Aerobus

v1.2

Generated by Doxygen 1.9.8

Chapter 1

Concept Index

1.1 Concepts

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

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::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

5.2 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

5.3 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

5.4 aerobus::ContinuedFraction< values > Struct Template Reference

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

`#include <lib.h>`

5.4.1 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

5.5 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

5.6 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

5.7 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

5.7.1 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

5.8 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 positive

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

5.8.1 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

5.9 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

5.10 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

5.11 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

5.11.1 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

5.12 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

5.12.1 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

5.12.2 Member Typedef Documentation

5.12.2.1 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	

5.12.2.2 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	

5.12.2.3 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	

5.12.2.4 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	

5.12.2.5 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	

5.12.2.6 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	

5.12.2.7 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	

5.12.2.8 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	

5.12.2.9 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	

5.12.2.10 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	

5.12.2.11 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	

5.12.2.12 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	

5.12.2.13 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

5.13 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

5.14 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

5.15 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

5.16 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)

5.16.1 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

5.17 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

5.18 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

5.18.1 Detailed Description

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

values in i32

Template Parameters

x	an actual integer

5.18.2 Member Function Documentation

5.18.2.1 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

5.18.2.2 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

5.19 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

5.19.1 Detailed Description

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

values in i64

Template Parameters

x	an actual integer

5.19.2 Member Function Documentation

5.19.2.1 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)

5.19.2.2 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

5.20 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

5.20.1 Member Typedef Documentation

5.20.1.1 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	

5.20.2 Member Function Documentation

5.20.2.1 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)

5.20.2.2 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

5.21 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

5.22 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

5.23 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

5.24 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

5.24.1 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

```
00001 // -*- lsst-c++ -*-
00002
00003 #include <cstdint> // NOLINT(clang-diagnostic-pragma-pack)
00004 #include <cstddef>
00005 #include <cstring>
00006 #include <type_traits>
00007 #include <utility>
00008 #include <algorithm>
00009 #include <functional>
00010 #include <string>
00011 #include <concepts>
00012 #include <array>
00013
00014
00015 #ifdef _MSC_VER
00016 #define ALIGNED(x) __declspec(align(x))
00017 #define INLINED __forceinline
00018 #else
00019 #define ALIGNED(x) __attribute__((aligned(x)))
00020 #define INLINED __attribute__((always_inline)) inline
00021 #endif
00022
00023 // aligned allocation
00024 namespace aerobus {
00031     template<typename T>
00032     T* aligned_malloc(size_t count, size_t alignment) {
00033         #ifdef _MSC_VER
00034         return static_cast<T*>(_aligned_malloc(count * sizeof(T), alignment));
00035         #else
00036         return static_cast<T*>(aligned_alloc(alignment, count * sizeof(T)));
00037         #endif
00038     }
00039
00040     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 } };
00041
00050     template<typename T, size_t N>
00051     constexpr bool contains(const std::array<T, N>& arr, const T& v) {
00052         for (const auto& vv : arr) {
00053             if (v == vv) {
00054                 return true;
00055             }
00056         }
00057
00058         return false;
00059     }
00060
00061 }
00062
00063 // concepts
00064 namespace aerobus
00065 {
00067     template <typename R>
00068     concept IsRing = requires {
00069         typename R::one;
00070         typename R::zero;
00071         typename R::template add_t<typename R::one, typename R::one>;
00072         typename R::template sub_t<typename R::one, typename R::one>;
00073         typename R::template mul_t<typename R::one, typename R::one>;
00074     };
00075
00077     template <typename R>
00078     concept IsEuclideanDomain = IsRing<R> && requires {
00079         typename R::template div_t<typename R::one, typename R::one>;
00080         typename R::template mod_t<typename R::one, typename R::one>;
00081         typename R::template gcd_t<typename R::one, typename R::one>;
00082         typename R::template eq_t<typename R::one, typename R::one>;
00083         typename R::template pos_t<typename R::one>;
00084
00085         R::template pos_v<typename R::one> == true;
00086         //typename R::template gt_t<typename R::one, typename R::zero>;
00087         R::is_euclidean_domain == true;
00088     };
00089
00091     template<typename R>
00092     concept IsField = IsEuclideanDomain<R> && requires {
00093         R::is_field == true;
00094     };
00095 }
00096
00097 // utilities
00098 namespace aerobus {
00099     namespace internal
00100     {
00101         template<template<typename...> typename TT, typename T>
00102         struct is_instantiation_of : std::false_type { };
00103
00104         template<template<typename...> typename TT, typename... Ts>
00105         struct is_instantiation_of<TT, TT<Ts...>> : std::true_type { };
```

```
00106
00107         template<template<typename...> typename TT, typename T>
00108         inline constexpr bool is_instantiation_of_v = is_instantiation_of<TT, T>::value;
00109
00110         template <int64_t i, typename T, typename... Ts>
00111         struct type_at
00112         {
00113             static_assert(i < sizeof...(Ts) + 1, "index out of range");
00114             using type = typename type_at<i - 1, Ts...>::type;
00115         };
00116
00117         template <typename T, typename... Ts> struct type_at<0, T, Ts...> {
00118             using type = T;
00119         };
00120
00121         template <size_t i, typename... Ts>
00122         using type_at_t = typename type_at<i, Ts...>::type;
00123
00124
00125         template<int32_t n, int32_t i, typename E = void>
00126         struct _is_prime {};
00127
00128         // first 1000 primes are precomputed and stored in a table
00129         template<int32_t n, int32_t i>
00130         struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (contains<int32_t, 1000>(primes, n))» :
      std::true_type {};
00131
00132         // first 1000 primes are precomputed and stored in a table
00133         template<int32_t n, int32_t i>
00134         struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (!contains<int32_t, 1000>(primes, n))» :
      std::false_type {};
00135
00136         template<int32_t n, int32_t i>
00137         struct _is_prime<n, i, std::enable_if_t<
00138             (n >= 7920) &&
00139             (i >= 5 && i * i <= n) &&
00140             (n % i == 0 || n % (i + 2) == 0)» : std::false_type {};
00141
00142
00143         template<int32_t n, int32_t i>
00144         struct _is_prime<n, i, std::enable_if_t<
00145             (n >= 7920) &&
00146             (i >= 5 && i * i <= n) &&
00147             (n % i != 0 && n % (i + 2) != 0)» {
00148             static constexpr bool value = _is_prime<n, i + 6>::value;
00149         };
00150
00151         template<int32_t n, int32_t i>
00152         struct _is_prime<n, i, std::enable_if_t<
00153             (n >= 7920) &&
00154             (i >= 5 && i * i > n)» : std::true_type {};
00155     }
00156
00159     template<int32_t n>
00160     struct is_prime {
00162         static constexpr bool value = internal::_is_prime<n, 5>::value;
00163     };
00164
00165     namespace internal {
00166         template <std::size_t... Is>
00167         constexpr auto index_sequence_reverse(std::index_sequence<Is...> const&)
00168             -> decltype(std::index_sequence<sizeof...(Is) - 1U - Is...>{});
00169
00170         template <std::size_t N>
00171         using make_index_sequence_reverse
00172             = decltype(index_sequence_reverse(std::make_index_sequence<N>{}));
00173
00179         template<typename Ring, typename E = void>
00180         struct gcd;
00181
00182         template<typename Ring>
00183         struct gcd<Ring, std::enable_if_t<Ring::is_euclidean_domain» {
00184             template<typename A, typename B, typename E = void>
00185             struct gcd_helper {};
00186
00187             // B = 0, A > 0
00188             template<typename A, typename B>
00189             struct gcd_helper<A, B, std::enable_if_t<
00190                 ((B::is_zero_t::value) &&
00191                     (Ring::template gt_t<A, typename Ring::zero>::value))»
00192             {
00193                 using type = A;
00194             };
00195
00196             // B = 0, A < 0
00197             template<typename A, typename B>
00198             struct gcd_helper<A, B, std::enable_if_t<
```

```
00199                 ((B::is_zero_t::value) &&
00200                     !(Ring::template gt_t<A, typename Ring::zero>::value))»
00201             {
00202                 using type = typename Ring::template sub_t<typename Ring::zero, A>;
00203             };
00204
00205             // B != 0
00206             template<typename A, typename B>
00207             struct gcd_helper<A, B, std::enable_if_t<
00208                 (!B::is_zero_t::value)
00209                 » {
00210             private:
00211                 // A / B
00212                 using k = typename Ring::template div_t<A, B>;
00213                 // A - (A/B)*B = A % B
00214                 using m = typename Ring::template sub_t<A, typename Ring::template mul_t<k, B»;
00215             public:
00216                 using type = typename gcd_helper<B, m>::type;
00217             };
00218
00219             template<typename A, typename B>
00220             using type = typename gcd_helper<A, B>::type;
00221         };
00222     }
00223
00226     template<typename T, typename A, typename B>
00227     using gcd_t = typename internal::gcd<T>::template type<A, B>;
00228 }
00229
00230 // quotient ring by the principal ideal generated by X
00231 namespace aerobus {
00232     template<typename Ring, typename X>
00233     requires IsRing<Ring>
00234     struct Quotient {
00235         template <typename V>
00236         struct val {
00237         private:
00238             using tmp = typename Ring::template mod_t<V, X>;
00239         public:
00240             using type = std::conditional_t<
00241                 Ring::template pos_v<tmp>,
00242                 tmp,
00243                 typename Ring::template sub_t<typename Ring::zero, tmp>
00244             >;
00245         };
00246
00247         using zero = val<typename Ring::zero>;
00248         using one = val<typename Ring::one>;
00249
00250         template<typename v1, typename v2>
00251         using add_t = val<typename Ring::template add_t<typename v1::type, typename v2::type>>;
00252         template<typename v1, typename v2>
00253         using mul_t = val<typename Ring::template mul_t<typename v1::type, typename v2::type>>;
00254         template<typename v1, typename v2>
00255         using div_t = val<typename Ring::template div_t<typename v1::type, typename v2::type>>;
00256         template<typename v1, typename v2>
00257         using mod_t = val<typename Ring::template mod_t<typename v1::type, typename v2::type>>;
00258         template<typename v1, typename v2>
00259         using eq_t = typename Ring::template eq_t<typename v1::type, typename v2::type>;
00260         template<typename v1>
00261         using pos_t = std::true_type;
00262
00263         template<typename v>
00264         static constexpr bool pos_v = pos_t<v>::value;
00265
00266         static constexpr bool is_euclidean_domain = true;
00267
00268         template<auto x>
00269         using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;
00270
00271         template<typename v>
00272         using inject_ring_t = val<v>;
00273     };
00274 }
00275
00276 // type_list
00277 namespace aerobus
00278 {
00280     template <typename... Ts>
00281     struct type_list;
00282
00283     namespace internal
00284     {
00285         template <typename T, typename... Us>
00286         struct pop_front_h
00287         {
00288             using tail = type_list<Us...>;
```

```
00289             using head = T;
00290         };
00291 
00292         template <uint64_t index, typename L1, typename L2>
00293         struct split_h
00294         {
00295         private:
00296             static_assert(index <= L2::length, "index ouf of bounds");
00297             using a = typename L2::pop_front::type;
00298             using b = typename L2::pop_front::tail;
00299             using c = typename L1::template push_back<a>;
00300 
00301         public:
00302             using head = typename split_h<index - 1, c, b>::head;
00303             using tail = typename split_h<index - 1, c, b>::tail;
00304         };
00305 
00306         template <typename L1, typename L2>
00307         struct split_h<0, L1, L2>
00308         {
00309             using head = L1;
00310             using tail = L2;
00311         };
00312 
00313         template <uint64_t index, typename L, typename T>
00314         struct insert_h
00315         {
00316             static_assert(index <= L::length, "index ouf of bounds");
00317             using s = typename L::template split<index>;
00318             using left = typename s::head;
00319             using right = typename s::tail;
00320             using ll = typename left::template push_back<T>;
00321             using type = typename ll::template concat<right>;
00322         };
00323 
00324         template <uint64_t index, typename L>
00325         struct remove_h
00326         {
00327             using s = typename L::template split<index>;
00328             using left = typename s::head;
00329             using right = typename s::tail;
00330             using rr = typename right::pop_front::tail;
00331             using type = typename left::template concat<rr>;
00332         };
00333     }
00334 
00335     template <typename... Ts>
00336     struct type_list
00337     {
00338     private:
00339         template <typename T>
00340         struct concat_h;
00341 
00342         template <typename... Us>
00343         struct concat_h<type_list<Us...»
00344         {
00345             using type = type_list<Ts..., Us...>;
00346         };
00347 
00348     public:
00349         static constexpr size_t length = sizeof...(Ts);
00350 
00351         template <typename T>
00352         using push_front = type_list<T, Ts...>;
00353 
00354         template <uint64_t index>
00355         using at = internal::type_at_t<index, Ts...>;
00356 
00357         struct pop_front
00358         {
00359             using type = typename internal::pop_front_h<Ts...>::head;
00360             using tail = typename internal::pop_front_h<Ts...>::tail;
00361         };
00362 
00363         template <typename T>
00364         using push_back = type_list<Ts..., T>;
00365 
00366         template <typename U>
00367         using concat = typename concat_h<U>::type;
00368 
00369         template <uint64_t index>
00370         struct split
00371         {
00372         private:
00373             using inner = internal::split_h<index, type_list<>, type_list<Ts...»;
00374 
00375         public:
```

```
00376             using head = typename inner::head;
00377             using tail = typename inner::tail;
00378         };
00379 
00380         template <uint64_t index, typename T>
00381         using insert = typename internal::insert_h<index, type_list<Ts...>, T>::type;
00382 
00383         template <uint64_t index>
00384         using remove = typename internal::remove_h<index, type_list<Ts...»::type;
00385     };
00386 
00387     template <>
00388     struct type_list<>
00389     {
00390         static constexpr size_t length = 0;
00391 
00392         template <typename T>
00393         using push_front = type_list<T>;
00394 
00395         template <typename T>
00396         using push_back = type_list<T>;
00397 
00398         template <typename U>
00399         using concat = U;
00400 
00401         // TODO: assert index == 0
00402         template <uint64_t index, typename T>
00403         using insert = type_list<T>;
00404     };
00405 }
00406 
00407 // i32
00408 namespace aerobus {
00410     struct i32 {
00411         using inner_type = int32_t;
00414         template<int32_t x>
00415         struct val {
00416             static constexpr int32_t v = x;
00417 
00420             template<typename valueType>
00421             static constexpr valueType get() { return static_cast<valueType>(x); }
00422 
00424             using is_zero_t = std::bool_constant<x == 0>;
00425 
00427             static std::string to_string() {
00428                 return std::to_string(x);
00429             }
00430 
00433             template<typename valueRing>
00434             static constexpr valueRing eval(const valueRing& v) {
00435                 return static_cast<valueRing>(x);
00436             }
00437         };
00438 
00440         using zero = val<0>;
00442         using one = val<1>;
00444         static constexpr bool is_field = false;
00446         static constexpr bool is_euclidean_domain = true;
00450         template<auto x>
00451         using inject_constant_t = val<static_cast<int32_t>(x)>;
00452 
00453         template<typename v>
00454         using inject_ring_t = v;
00455 
00456     private:
00457         template<typename v1, typename v2>
00458         struct add {
00459             using type = val<v1::v + v2::v>;
00460         };
00461 
00462         template<typename v1, typename v2>
00463         struct sub {
00464             using type = val<v1::v - v2::v>;
00465         };
00466 
00467         template<typename v1, typename v2>
00468         struct mul {
00469             using type = val<v1::v* v2::v>;
00470         };
00471 
00472         template<typename v1, typename v2>
00473         struct div {
00474             using type = val<v1::v / v2::v>;
00475         };
00476 
00477         template<typename v1, typename v2>
00478         struct remainder {
```

```
00479             using type = val<v1::v % v2::v>;
00480         };
00481
00482         template<typename v1, typename v2>
00483         struct gt {
00484             using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
00485         };
00486
00487         template<typename v1, typename v2>
00488         struct lt {
00489             using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
00490         };
00491
00492         template<typename v1, typename v2>
00493         struct eq {
00494             using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
00495         };
00496
00497         template<typename v1>
00498         struct pos {
00499             using type = std::bool_constant<(v1::v > 0)>;
00500         };
00501
00502     public:
00504         template<typename v1, typename v2>
00505         using add_t = typename add<v1, v2>::type;
00506
00508         template<typename v1, typename v2>
00509         using sub_t = typename sub<v1, v2>::type;
00510
00512         template<typename v1, typename v2>
00513         using mul_t = typename mul<v1, v2>::type;
00514
00516         template<typename v1, typename v2>
00517         using div_t = typename div<v1, v2>::type;
00518
00520         template<typename v1, typename v2>
00521         using mod_t = typename remainder<v1, v2>::type;
00522
00524         template<typename v1, typename v2>
00525         using gt_t = typename gt<v1, v2>::type;
00526
00528         template<typename v1, typename v2>
00529         using lt_t = typename lt<v1, v2>::type;
00530
00532         template<typename v1, typename v2>
00533         using eq_t = typename eq<v1, v2>::type;
00534
00536         template<typename v1, typename v2>
00537         using gcd_t = gcd_t<i32, v1, v2>;
00538
00540         template<typename v>
00541         using pos_t = typename pos<v>::type;
00542
00543         template<typename v>
00544         static constexpr bool pos_v = pos_t<v>::value;
00545     };
00546 }
00547
00548 // i64
00549 namespace aerobus {
00551     struct i64 {
00552         using inner_type = int64_t;
00555         template<int64_t x>
00556         struct val {
00557             static constexpr int64_t v = x;
00558
00561             template<typename valueType>
00562             static constexpr valueType get() { return static_cast<valueType>(x); }
00563
00565             using is_zero_t = std::bool_constant<x == 0>;
00566
00568             static std::string to_string() {
00569                 return std::to_string(x);
00570             }
00571
00574             template<typename valueRing>
00575             static constexpr valueRing eval(const valueRing& v) {
00576                 return static_cast<valueRing>(x);
00577             }
00578         };
00579
00583         template<auto x>
00584         using inject_constant_t = val<static_cast<int64_t>(x)>;
00585
00586         template<typename v>
00587         using inject_ring_t = v;
```

```
00588
00590         using zero = val<0>;
00592         using one = val<1>;
00594         static constexpr bool is_field = false;
00596         static constexpr bool is_euclidean_domain = true;
00597
00598     private:
00599         template<typename v1, typename v2>
00600         struct add {
00601             using type = val<v1::v + v2::v>;
00602         };
00603
00604         template<typename v1, typename v2>
00605         struct sub {
00606             using type = val<v1::v - v2::v>;
00607         };
00608
00609         template<typename v1, typename v2>
00610         struct mul {
00611             using type = val<v1::v* v2::v>;
00612         };
00613
00614         template<typename v1, typename v2>
00615         struct div {
00616             using type = val<v1::v / v2::v>;
00617         };
00618
00619         template<typename v1, typename v2>
00620         struct remainder {
00621             using type = val<v1::v% v2::v>;
00622         };
00623
00624         template<typename v1, typename v2>
00625         struct gt {
00626             using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
00627         };
00628
00629         template<typename v1, typename v2>
00630         struct lt {
00631             using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
00632         };
00633
00634         template<typename v1, typename v2>
00635         struct eq {
00636             using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
00637         };
00638
00639         template<typename v>
00640         struct pos {
00641             using type = std::bool_constant<(v::v > 0)>;
00642         };
00643
00644     public:
00646         template<typename v1, typename v2>
00647         using add_t = typename add<v1, v2>::type;
00648
00650         template<typename v1, typename v2>
00651         using sub_t = typename sub<v1, v2>::type;
00652
00654         template<typename v1, typename v2>
00655         using mul_t = typename mul<v1, v2>::type;
00656
00658         template<typename v1, typename v2>
00659         using div_t = typename div<v1, v2>::type;
00660
00662         template<typename v1, typename v2>
00663         using mod_t = typename remainder<v1, v2>::type;
00664
00666         template<typename v1, typename v2>
00667         using gt_t = typename gt<v1, v2>::type;
00668
00670         template<typename v1, typename v2>
00671         using lt_t = typename lt<v1, v2>::type;
00672
00674         template<typename v1, typename v2>
00675         using eq_t = typename eq<v1, v2>::type;
00676
00678         template<typename v1, typename v2>
00679         using gcd_t = gcd_t<i64, v1, v2>;
00680
00682         template<typename v>
00683         using pos_t = typename pos<v>::type;
00684
00685         template<typename v>
00686         static constexpr bool pos_v = pos_t<v>::value;
00687     };
00688 }
```

```
00689
00690 // z/pz
00691 namespace aerobus {
00696     template<int32_t p>
00697     struct zpz {
00698         using inner_type = int32_t;
00699         template<int32_t x>
00700         struct val {
00701             static constexpr int32_t v = x % p;
00702
00703             template<typename valueType>
00704             static constexpr valueType get() { return static_cast<valueType>(x % p); }
00705
00706             using is_zero_t = std::bool_constant<x% p == 0>;
00707             static std::string to_string() {
00708                 return std::to_string(x % p);
00709             }
00710
00711             template<typename valueRing>
00712             static constexpr valueRing eval(const valueRing& v) {
00713                 return static_cast<valueRing>(x % p);
00714             }
00715         };
00716
00717         template<auto x>
00718         using inject_constant_t = val<static_cast<int32_t>(x)>;
00719
00720         using zero = val<0>;
00721         using one = val<1>;
00722         static constexpr bool is_field = is_prime<p>::value;
00723         static constexpr bool is_euclidean_domain = true;
00724
00725     private:
00726         template<typename v1, typename v2>
00727         struct add {
00728             using type = val<(v1::v + v2::v) % p>;
00729         };
00730
00731         template<typename v1, typename v2>
00732         struct sub {
00733             using type = val<(v1::v - v2::v) % p>;
00734         };
00735
00736         template<typename v1, typename v2>
00737         struct mul {
00738             using type = val<(v1::v* v2::v) % p>;
00739         };
00740
00741         template<typename v1, typename v2>
00742         struct div {
00743             using type = val<(v1::v% p) / (v2::v % p)>;
00744         };
00745
00746         template<typename v1, typename v2>
00747         struct remainder {
00748             using type = val<(v1::v% v2::v) % p>;
00749         };
00750
00751         template<typename v1, typename v2>
00752         struct gt {
00753             using type = std::conditional_t<(v1::v% p > v2::v% p), std::true_type, std::false_type>;
00754         };
00755
00756         template<typename v1, typename v2>
00757         struct lt {
00758             using type = std::conditional_t<(v1::v% p < v2::v% p), std::true_type, std::false_type>;
00759         };
00760
00761         template<typename v1, typename v2>
00762         struct eq {
00763             using type = std::conditional_t<(v1::v% p == v2::v % p), std::true_type, std::false_type>;
00764         };
00765
00766         template<typename v1>
00767         struct pos {
00768             using type = std::bool_constant<(v1::v > 0)>;
00769         };
00770
00771     public:
00772         template<typename v1, typename v2>
00773         using add_t = typename add<v1, v2>::type;
00774
00775         template<typename v1, typename v2>
00776         using sub_t = typename sub<v1, v2>::type;
00777
00778         template<typename v1, typename v2>
00779         using mul_t = typename mul<v1, v2>::type;
```

```
00780
00781         template<typename v1, typename v2>
00782         using div_t = typename div<v1, v2>::type;
00783
00784         template<typename v1, typename v2>
00785         using mod_t = typename remainder<v1, v2>::type;
00786
00787         template<typename v1, typename v2>
00788         using gt_t = typename gt<v1, v2>::type;
00789
00790         template<typename v1, typename v2>
00791         using lt_t = typename lt<v1, v2>::type;
00792
00793         template<typename v1, typename v2>
00794         using eq_t = typename eq<v1, v2>::type;
00795
00796         template<typename v1, typename v2>
00797         using gcd_t = gcd_t<i32, v1, v2>;
00798
00799         template<typename v1>
00800         using pos_t = typename pos<v1>::type;
00801
00802         template<typename v>
00803         static constexpr bool pos_v = pos_t<v>::value;
00804     };
00805 }
00806
00807 // polynomial
00808 namespace aerobus {
00809     // coeffN x^N + ...
00814     template<typename Ring, char variable_name = 'x'>
00815     requires IsEuclideanDomain<Ring>
00816     struct polynomial {
00817         static constexpr bool is_field = false;
00818         static constexpr bool is_euclidean_domain = Ring::is_euclidean_domain;
00819
00820         template<typename coeffN, typename... coeffs>
00821         struct val {
00823             static constexpr size_t degree = sizeof...(coeffs);
00825             using aN = coeffN;
00827             using strip = val<coeffs...>;
00829             using is_zero_t = std::bool_constant<(degree == 0) && (aN::is_zero_t::value)>;
00830
00831             private:
00832             template<size_t index, typename E = void>
00833             struct coeff_at {};
00834
00835             template<size_t index>
00836             struct coeff_at<index, std::enable_if_t<(index >= 0 && index <= sizeof...(coeffs))>> {
00837                 using type = internal::type_at_t<sizeof...(coeffs) - index, coeffN, coeffs...>;
00838             };
00839
00840             template<size_t index>
00841             struct coeff_at<index, std::enable_if_t<(index < 0 || index > sizeof...(coeffs))>> {
00842                 using type = typename Ring::zero;
00843             };
00844
00845             public:
00848             template<size_t index>
00849             using coeff_at_t = typename coeff_at<index>::type;
00850
00853             static std::string to_string() {
00854                 return string_helper<coeffN, coeffs...>::func();
00855             }
00856
00861             template<typename valueRing>
00862             static constexpr valueRing eval(const valueRing& x) {
00863                 return eval_helper<valueRing, val>::template inner<0, degree +
      1>::func(static_cast<valueRing>(0), x);
00864             }
00865         };
00866
00867         // specialization for constants
00868         template<typename coeffN>
00869         struct val<coeffN> {
00870             static constexpr size_t degree = 0;
00871             using aN = coeffN;
00872             using strip = val<coeffN>;
00873             using is_zero_t = std::bool_constant<aN::is_zero_t::value>;
00874
00875             template<size_t index, typename E = void>
00876             struct coeff_at {};
00877
00878             template<size_t index>
00879             struct coeff_at<index, std::enable_if_t<(index == 0)>> {
00880                 using type = aN;
00881             };
```

```
00882
00883             template<size_t index>
00884             struct coeff_at<index, std::enable_if_t<(index < 0 || index > 0)» {
00885                 using type = typename Ring::zero;
00886             };
00887
00888             template<size_t index>
00889             using coeff_at_t = typename coeff_at<index>::type;
00890
00891             static std::string to_string() {
00892                 return string_helper<coeffN>::func();
00893             }
00894
00895             template<typename valueRing>
00896             static constexpr valueRing eval(const valueRing& x) {
00897                 return static_cast<valueRing>(aN::template get<valueRing>());
00898             }
00899         };
00900
00902         using zero = val<typename Ring::zero>;
00904         using one = val<typename Ring::one>;
00906         using X = val<typename Ring::one, typename Ring::zero>;
00907
00908     private:
00909         template<typename P, typename E = void>
00910         struct simplify;
00911
00912         template <typename P1, typename P2, typename I>
00913         struct add_low;
00914
00915         template<typename P1, typename P2>
00916         struct add {
00917             using type = typename simplify<typename add_low<
00918             P1,
00919             P2,
00920             internal::make_index_sequence_reverse<
00921             std::max(P1::degree, P2::degree) + 1
00922             »::type>::type;
00923         };
00924
00925         template <typename P1, typename P2, typename I>
00926         struct sub_low;
00927
00928         template <typename P1, typename P2, typename I>
00929         struct mul_low;
00930
00931         template<typename v1, typename v2>
00932         struct mul {
00933                 using type = typename mul_low<
00934                     v1,
00935                     v2,
00936                     internal::make_index_sequence_reverse<
00937                     v1::degree + v2::degree + 1
00938                     »::type;
00939         };
00940
00941         template<typename coeff, size_t deg>
00942         struct monomial;
00943
00944         template<typename v, typename E = void>
00945         struct derive_helper {};
00946
00947         template<typename v>
00948         struct derive_helper<v, std::enable_if_t<v::degree == 0» {
00949             using type = zero;
00950         };
00951
00952         template<typename v>
00953         struct derive_helper<v, std::enable_if_t<v::degree != 0» {
00954             using type = typename add<
00955                 typename derive_helper<typename simplify<typename v::strip>::type>::type,
00956                 typename monomial<
00957                     typename Ring::template mul_t<
00958                         typename v::aN,
00959                         typename Ring::template inject_constant_t<(v::degree)>
00960                     >,
00961                     v::degree - 1
00962                 >::type
00963             >::type;
00964         };
00965
00966         template<typename v1, typename v2, typename E = void>
00967         struct eq_helper {};
00968
00969         template<typename v1, typename v2>
00970         struct eq_helper<v1, v2, std::enable_if_t<v1::degree != v2::degree» {
00971             using type = std::false_type;
```

```
00972         };
00973
00974
00975         template<typename v1, typename v2>
00976         struct eq_helper<v1, v2, std::enable_if_t<
00977             v1::degree == v2::degree &&
00978             (v1::degree != 0 || v2::degree != 0) &&
00979             std::is_same<
00980             typename Ring::template eq_t<typename v1::aN, typename v2::aN>,
00981             std::false_type
00982             >::value
00983         >
00984         > {
00985             using type = std::false_type;
00986         };
00987
00988         template<typename v1, typename v2>
00989         struct eq_helper<v1, v2, std::enable_if_t<
00990             v1::degree == v2::degree &&
00991             (v1::degree != 0 || v2::degree != 0) &&
00992             std::is_same<
00993             typename Ring::template eq_t<typename v1::aN, typename v2::aN>,
00994             std::true_type
00995             >::value
00996         >» {
00997             using type = typename eq_helper<typename v1::strip, typename v2::strip>::type;
00998         };
00999
01000         template<typename v1, typename v2>
01001         struct eq_helper<v1, v2, std::enable_if_t<
01002             v1::degree == v2::degree &&
01003             (v1::degree == 0)
01004         >» {
01005             using type = typename Ring::template eq_t<typename v1::aN, typename v2::aN>;
01006         };
01007
01008         template<typename v1, typename v2, typename E = void>
01009         struct lt_helper {};
01010
01011         template<typename v1, typename v2>
01012         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)>» {
01013             using type = std::true_type;
01014         };
01015
01016         template<typename v1, typename v2>
01017         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree == v2::degree)>» {
01018             using type = typename Ring::template lt_t<typename v1::aN, typename v2::aN>;
01019         };
01020
01021         template<typename v1, typename v2>
01022         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)>» {
01023             using type = std::false_type;
01024         };
01025
01026         template<typename v1, typename v2, typename E = void>
01027         struct gt_helper {};
01028
01029         template<typename v1, typename v2>
01030         struct gt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)>» {
01031             using type = std::true_type;
01032         };
01033
01034         template<typename v1, typename v2>
01035         struct gt_helper<v1, v2, std::enable_if_t<(v1::degree == v2::degree)>» {
01036             using type = std::false_type;
01037         };
01038
01039         template<typename v1, typename v2>
01040         struct gt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)>» {
01041             using type = std::false_type;
01042         };
01043
01044         // when high power is zero : strip
01045         template<typename P>
01046         struct simplify<P, std::enable_if_t<
01047             std::is_same<
01048             typename Ring::zero,
01049             typename P::aN
01050             >::value && (P::degree > 0)
01051         >»
01052         {
01053             using type = typename simplify<typename P::strip>::type;
01054         };
01055
01056         // otherwise : do nothing
01057         template<typename P>
01058         struct simplify<P, std::enable_if_t<
```

```
01059             !std::is_same<
01060             typename Ring::zero,
01061             typename P::aN
01062             >::value && (P::degree > 0)
01063         »
01064         {
01065             using type = P;
01066         };
01067 
01068         // do not simplify constants
01069         template<typename P>
01070         struct simplify<P, std::enable_if_t<P::degree == 0» {
01071             using type = P;
01072         };
01073 
01074         // addition at
01075         template<typename P1, typename P2, size_t index>
01076         struct add_at {
01077             using type =
01078                 typename Ring::template add_t<typename P1::template coeff_at_t<index>, typename
      P2::template coeff_at_t<index>>;
01079         };
01080 
01081         template<typename P1, typename P2, size_t index>
01082         using add_at_t = typename add_at<P1, P2, index>::type;
01083 
01084         template<typename P1, typename P2, std::size_t... I>
01085         struct add_low<P1, P2, std::index_sequence<I...» {
01086             using type = val<add_at_t<P1, P2, I>...>;
01087         };
01088 
01089         // substraction at
01090         template<typename P1, typename P2, size_t index>
01091         struct sub_at {
01092             using type =
01093                 typename Ring::template sub_t<typename P1::template coeff_at_t<index>, typename
      P2::template coeff_at_t<index>>;
01094         };
01095 
01096         template<typename P1, typename P2, size_t index>
01097         using sub_at_t = typename sub_at<P1, P2, index>::type;
01098 
01099         template<typename P1, typename P2, std::size_t... I>
01100         struct sub_low<P1, P2, std::index_sequence<I...» {
01101             using type = val<sub_at_t<P1, P2, I>...>;
01102         };
01103 
01104         template<typename P1, typename P2>
01105         struct sub {
01106             using type = typename simplify<typename sub_low<
01107             P1,
01108             P2,
01109             internal::make_index_sequence_reverse<
01110             std::max(P1::degree, P2::degree) + 1
01111             »::type>::type;
01112         };
01113 
01114         // multiplication at
01115         template<typename v1, typename v2, size_t k, size_t index, size_t stop>
01116         struct mul_at_loop_helper {
01117             using type = typename Ring::template add_t<
01118                 typename Ring::template mul_t<
01119                 typename v1::template coeff_at_t<index>,
01120                 typename v2::template coeff_at_t<k - index>
01121                 >,
01122                 typename mul_at_loop_helper<v1, v2, k, index + 1, stop>::type
01123             >;
01124         };
01125 
01126         template<typename v1, typename v2, size_t k, size_t stop>
01127         struct mul_at_loop_helper<v1, v2, k, stop, stop> {
01128             using type = typename Ring::template mul_t<typename v1::template coeff_at_t<stop>,
      typename v2::template coeff_at_t<0>>;
01129         };
01130 
01131         template <typename v1, typename v2, size_t k, typename E = void>
01132         struct mul_at {};
01133 
01134         template<typename v1, typename v2, size_t k>
01135         struct mul_at<v1, v2, k, std::enable_if_t<(k < 0) || (k > v1::degree + v2::degree)» {
01136             using type = typename Ring::zero;
01137         };
01138 
01139         template<typename v1, typename v2, size_t k>
01140         struct mul_at<v1, v2, k, std::enable_if_t<(k >= 0) && (k <= v1::degree + v2::degree)» {
01141             using type = typename mul_at_loop_helper<v1, v2, k, 0, k>::type;
01142         };
```

```
01143
01144         template<typename P1, typename P2, size_t index>
01145         using mul_at_t = typename mul_at<P1, P2, index>::type;
01146
01147         template<typename P1, typename P2, std::size_t... I>
01148         struct mul_low<P1, P2, std::index_sequence<I...>> {
01149             using type = val<mul_at_t<P1, P2, I>...>;
01150         };
01151
01152         // division helper
01153         template< typename A, typename B, typename Q, typename R, typename E = void>
01154         struct div_helper {};
01155
01156         template<typename A, typename B, typename Q, typename R>
01157         struct div_helper<A, B, Q, R, std::enable_if_t<
01158             (R::degree < B::degree) ||
01159             (R::degree == 0 && std::is_same<typename R::aN, typename Ring::zero>::value)>> {
01160             using q_type = Q;
01161             using mod_type = R;
01162             using gcd_type = B;
01163         };
01164
01165         template<typename A, typename B, typename Q, typename R>
01166         struct div_helper<A, B, Q, R, std::enable_if_t<
01167             (R::degree >= B::degree) &&
01168             !(R::degree == 0 && std::is_same<typename R::aN, typename Ring::zero>::value)>> {
01169         private:
01170             using rN = typename R::aN;
01171             using bN = typename B::aN;
01172             using pT = typename monomial<typename Ring::template div_t<rN, bN>, R::degree -
      B::degree>::type;
01173             using rr = typename sub<R, typename mul<pT, B>::type>::type;
01174             using qq = typename add<Q, pT>::type;
01175
01176         public:
01177             using q_type = typename div_helper<A, B, qq, rr>::q_type;
01178             using mod_type = typename div_helper<A, B, qq, rr>::mod_type;
01179             using gcd_type = rr;
01180         };
01181
01182         template<typename A, typename B>
01183         struct div {
01184             static_assert(Ring::is_euclidean_domain, "cannot divide in that type of Ring");
01185             using q_type = typename div_helper<A, B, zero, A>::q_type;
01186             using m_type = typename div_helper<A, B, zero, A>::mod_type;
01187         };
01188
01189
01190         template<typename P>
01191         struct make_unit {
01192             using type = typename div<P, val<typename P::aN>>::q_type;
01193         };
01194
01195         template<typename coeff, size_t deg>
01196         struct monomial {
01197             using type = typename mul<X, typename monomial<coeff, deg - 1>::type>::type;
01198         };
01199
01200         template<typename coeff>
01201         struct monomial<coeff, 0> {
01202             using type = val<coeff>;
01203         };
01204
01205         template<typename valueRing, typename P>
01206         struct eval_helper
01207         {
01208             template<size_t index, size_t stop>
01209             struct inner {
01210                 static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
01211                     constexpr valueRing coeff = static_cast<valueRing>(P::template
      coeff_at_t<P::degree - index>::template get<valueRing>());
01212                     return eval_helper<valueRing, P>::template inner<index + 1, stop>::func(x * accum
      + coeff, x);
01213                 }
01214             };
01215
01216             template<size_t stop>
01217             struct inner<stop, stop> {
01218                 static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
01219                     return accum;
01220                 }
01221             };
01222         };
01223
01224         template<typename coeff, typename... coeffs>
01225         struct string_helper {
01226             static std::string func() {
```

```
01227                 std::string tail = string_helper<coeffs...>::func();
01228                 std::string result = "";
01229                 if (Ring::template eq_t<coeff, typename Ring::zero>::value) {
01230                     return tail;
01231                 }
01232                 else if (Ring::template eq_t<coeff, typename Ring::one>::value) {
01233                     if (sizeof...(coeffs) == 1) {
01234                         result += std::string(1, variable_name);
01235                     }
01236                     else {
01237                         result += std::string(1, variable_name) + "^" +
      std::to_string(sizeof...(coeffs));
01238                     }
01239                 }
01240                 else {
01241                     if (sizeof...(coeffs) == 1) {
01242                         result += coeff::to_string() + " " + std::string(1, variable_name);
01243                     }
01244                     else {
01245                         result += coeff::to_string() + " " + std::string(1, variable_name) + "^" +
      std::to_string(sizeof...(coeffs));
01246                     }
01247                 }
01248 
01249                 if(!tail.empty()) {
01250                     result += " + " + tail;
01251                 }
01252 
01253                 return result;
01254             }
01255         };
01256 
01257         template<typename coeff>
01258         struct string_helper<coeff> {
01259             static std::string func() {
01260                 if(!std::is_same<coeff, typename Ring::zero>::value) {
01261                     return coeff::to_string();
01262                 } else {
01263                     return "";
01264                 }
01265             }
01266         };
01267 
01268     public:
01271         template<typename P>
01272         using simplify_t = typename simplify<P>::type;
01273 
01277         template<typename v1, typename v2>
01278         using add_t = typename add<v1, v2>::type;
01279 
01283         template<typename v1, typename v2>
01284         using sub_t = typename sub<v1, v2>::type;
01285 
01289         template<typename v1, typename v2>
01290         using mul_t = typename mul<v1, v2>::type;
01291 
01295         template<typename v1, typename v2>
01296         using eq_t = typename eq_helper<v1, v2>::type;
01297 
01301         template<typename v1, typename v2>
01302         using lt_t = typename lt_helper<v1, v2>::type;
01303 
01307         template<typename v1, typename v2>
01308         using gt_t = typename gt_helper<v1, v2>::type;
01309 
01313         template<typename v1, typename v2>
01314         using div_t = typename div<v1, v2>::q_type;
01315 
01319         template<typename v1, typename v2>
01320         using mod_t = typename div_helper<v1, v2, zero, v1>::mod_type;
01321 
01325         template<typename coeff, size_t deg>
01326         using monomial_t = typename monomial<coeff, deg>::type;
01327 
01330         template<typename v>
01331         using derive_t = typename derive_helper<v>::type;
01332 
01335         template<typename v>
01336         using pos_t = typename Ring::template pos_t<typename v::aN>;
01337 
01338         template<typename v>
01339         static constexpr bool pos_v = pos_t<v>::value;
01340 
01344         template<typename v1, typename v2>
01345         using gcd_t = std::conditional_t<
01346             Ring::is_euclidean_domain,
01347             typename make_unit<gcd_t<polynomial<Ring, variable_name>, v1, v2»::type,
```

```
01348             void>;
01349
01353         template<auto x>
01354         using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;
01355
01359         template<typename v>
01360         using inject_ring_t = val<v>;
01361     };
01362 }
01363
01364 // fraction field
01365 namespace aerobus {
01366     namespace internal {
01367         template<typename Ring, typename E = void>
01368         requires IsEuclideanDomain<Ring>
01369         struct _FractionField {};
01370
01371         template<typename Ring>
01372         requires IsEuclideanDomain<Ring>
01373         struct _FractionField<Ring, std::enable_if_t<Ring::is_euclidean_domain»
01374         {
01376             static constexpr bool is_field = true;
01377             static constexpr bool is_euclidean_domain = true;
01378
01379             private:
01380             template<typename val1, typename val2, typename E = void>
01381             struct to_string_helper {};
01382
01383             template<typename val1, typename val2>
01384             struct to_string_helper <val1, val2,
01385                 std::enable_if_t<
01386                 Ring::template eq_t<
01387                 val2, typename Ring::one
01388                 >::value
01389                 >
01390             > {
01391                 static std::string func() {
01392                     return val1::to_string();
01393                 }
01394             };
01395
01396             template<typename val1, typename val2>
01397             struct to_string_helper<val1, val2,
01398                 std::enable_if_t<
01399                 !Ring::template eq_t<
01400                 val2,
01401                 typename Ring::one
01402                 >::value
01403                 >
01404             > {
01405                 static std::string func() {
01406                     return "(" + val1::to_string() + ") / (" + val2::to_string() + ")";
01407                 }
01408             };
01409
01410             public:
01414             template<typename val1, typename val2>
01415             struct val {
01416                 using x = val1;
01417                 using y = val2;
01419                 using is_zero_t = typename val1::is_zero_t;
01420                 using ring_type = Ring;
01421                 using field_type = _FractionField<Ring>;
01422
01424                 static constexpr bool is_integer = std::is_same<val2, typename Ring::one>::value;
01425
01429                 template<typename valueType>
01430                 static constexpr valueType get() { return static_cast<valueType>(x::v) /
      static_cast<valueType>(y::v); }
01431
01434                 static std::string to_string() {
01435                     return to_string_helper<val1, val2>::func();
01436                 }
01437
01442                 template<typename valueRing>
01443                 static constexpr valueRing eval(const valueRing& v) {
01444                     return x::eval(v) / y::eval(v);
01445                 }
01446             };
01447
01449             using zero = val<typename Ring::zero, typename Ring::one>;
01451             using one = val<typename Ring::one, typename Ring::one>;
01452
01455             template<typename v>
01456             using inject_t = val<v, typename Ring::one>;
01457
01460             template<auto x>
```

```
01461             using inject_constant_t = val<typename Ring::template inject_constant_t<x>, typename
      Ring::one>;
01462
01465             template<typename v>
01466             using inject_ring_t = val<typename Ring::template inject_ring_t<v>, typename Ring::one>;
01467
01468             using ring_type = Ring;
01469
01470         private:
01471             template<typename v, typename E = void>
01472             struct simplify {};
01473
01474             // x = 0
01475             template<typename v>
01476             struct simplify<v, std::enable_if_t<v::x::is_zero_t::value» {
01477                 using type = typename _FractionField<Ring>::zero;
01478             };
01479
01480             // x != 0
01481             template<typename v>
01482             struct simplify<v, std::enable_if_t<!v::x::is_zero_t::value» {
01483
01484             private:
01485                 using _gcd = typename Ring::template gcd_t<typename v::x, typename v::y>;
01486                 using newx = typename Ring::template div_t<typename v::x, _gcd>;
01487                 using newy = typename Ring::template div_t<typename v::y, _gcd>;
01488
01489                 using posx = std::conditional_t<!Ring::template pos_v<newy>, typename Ring::template
      sub_t<typename Ring::zero, newx>, newx>;
01490                 using posy = std::conditional_t<!Ring::template pos_v<newy>, typename Ring::template
      sub_t<typename Ring::zero, newy>, newy>;
01491             public:
01492                 using type = typename _FractionField<Ring>::template val<posx, posy>;
01493             };
01494
01495         public:
01498             template<typename v>
01499             using simplify_t = typename simplify<v>::type;
01500
01501         private:
01502
01503             template<typename v1, typename v2>
01504             struct add {
01505             private:
01506                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01507                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01508                 using dividend = typename Ring::template add_t<a, b>;
01509                 using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01510                 using g = typename Ring::template gcd_t<dividend, diviser>;
01511
01512             public:
01513                 using type = typename _FractionField<Ring>::template simplify_t<val<dividend,
      diviser»;
01514             };
01515
01516             template<typename v>
01517             struct pos {
01518                 using type = std::conditional_t<
01519                     (Ring::template pos_v<typename v::x> && Ring::template pos_v<typename v::y>) ||
01520                     (!Ring::template pos_v<typename v::x> && !Ring::template pos_v<typename v::y>),
01521                     std::true_type,
01522                     std::false_type>;
01523
01524             };
01525
01526             template<typename v1, typename v2>
01527             struct sub {
01528             private:
01529                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01530                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01531                 using dividend = typename Ring::template sub_t<a, b>;
01532                 using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01533                 using g = typename Ring::template gcd_t<dividend, diviser>;
01534
01535             public:
01536                 using type = typename _FractionField<Ring>::template simplify_t<val<dividend,
      diviser»;
01537             };
01538
01539             template<typename v1, typename v2>
01540             struct mul {
01541             private:
01542                 using a = typename Ring::template mul_t<typename v1::x, typename v2::x>;
01543                 using b = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01544
01545             public:
01546                 using type = typename _FractionField<Ring>::template simplify_t<val<a, b»;
```

```
01547             };
01548
01549             template<typename v1, typename v2, typename E = void>
01550             struct div {};
01551
01552             template<typename v1, typename v2>
01553             struct div<v1, v2, std::enable_if_t<!std::is_same<v2, typename
      _FractionField<Ring>::zero>::value>  {
01554             private:
01555                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01556                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01557
01558             public:
01559                 using type = typename _FractionField<Ring>::template simplify_t<val<a, b>;
01560             };
01561
01562             template<typename v1, typename v2>
01563             struct div<v1, v2, std::enable_if_t<
01564                 std::is_same<zero, v1>::value && std::is_same<v2, zero>::value>  {
01565                 using type = one;
01566             };
01567
01568             template<typename v1, typename v2>
01569             struct eq {
01570                 using type = std::conditional_t<
01571                         std::is_same<typename simplify_t<v1>::x, typename simplify_t<v2>::x>::value &&
01572                         std::is_same<typename simplify_t<v1>::y, typename simplify_t<v2>::y>::value,
01573                     std::true_type,
01574                     std::false_type>;
01575             };
01576
01577             template<typename TL, typename E = void>
01578             struct vadd {};
01579
01580             template<typename TL>
01581             struct vadd<TL, std::enable_if_t<(TL::length > 1)>> {
01582                 using head = typename TL::pop_front::type;
01583                 using tail = typename TL::pop_front::tail;
01584                 using type = typename add<head, typename vadd<tail>::type>::type;
01585             };
01586
01587             template<typename TL>
01588             struct vadd<TL, std::enable_if_t<(TL::length == 1)>> {
01589                 using type = typename TL::template at<0>;
01590             };
01591
01592             template<typename... vals>
01593             struct vmul {};
01594
01595             template<typename v1, typename... vals>
01596             struct vmul<v1, vals...> {
01597                 using type = typename mul<v1, typename vmul<vals...>::type>::type;
01598             };
01599
01600             template<typename v1>
01601             struct vmul<v1> {
01602                 using type = v1;
01603             };
01604
01605
01606             template<typename v1, typename v2, typename E = void>
01607             struct gt;
01608
01609             template<typename v1, typename v2>
01610             struct gt<v1, v2, std::enable_if_t<
01611                 (eq<v1, v2>::type::value)
01612                 >> {
01613                 using type = std::false_type;
01614             };
01615
01616             template<typename v1, typename v2>
01617             struct gt<v1, v2, std::enable_if_t<
01618                 (!eq<v1, v2>::type::value) &&
01619                 (!pos<v1>::type::value) && (!pos<v2>::type::value)
01620                 >> {
01621                 using type = typename gt<
01622                     typename sub<zero, v1>::type, typename sub<zero, v2>::type
01623                 >::type;
01624             };
01625
01626             template<typename v1, typename v2>
01627             struct gt<v1, v2, std::enable_if_t<
01628                 (!eq<v1, v2>::type::value) &&
01629                 (pos<v1>::type::value) && (!pos<v2>::type::value)
01630                 >> {
01631                 using type = std::true_type;
01632             };
```

```
01633
01634             template<typename v1, typename v2>
01635             struct gt<v1, v2, std::enable_if_t<
01636                 (!eq<v1, v2>::type::value) &&
01637                 (!pos<v1>::type::value) && (pos<v2>::type::value)
01638                 » {
01639                 using type = std::false_type;
01640             };
01641
01642             template<typename v1, typename v2>
01643             struct gt<v1, v2, std::enable_if_t<
01644                 (!eq<v1, v2>::type::value) &&
01645                 (pos<v1>::type::value) && (pos<v2>::type::value)
01646                 » {
01647                 using type = typename Ring::template gt_t<
01648                     typename Ring::template mul_t<v1::x, v2::y>,
01649                     typename Ring::template mul_t<v2::y, v2::x>
01650                 >;
01651             };
01652
01653
01654         public:
01656             template<typename v1, typename v2>
01657             using add_t = typename add<v1, v2>::type;
01659             template<typename v1, typename v2>
01660             using mod_t = zero;
01664             template<typename v1, typename v2>
01665             using gcd_t = v1;
01668             template<typename... vs>
01669             using vadd_t = typename vadd<vs...>::type;
01672             template<typename... vs>
01673             using vmul_t = typename vmul<vs...>::type;
01675             template<typename v1, typename v2>
01676             using sub_t = typename sub<v1, v2>::type;
01678             template<typename v1, typename v2>
01679             using mul_t = typename mul<v1, v2>::type;
01681             template<typename v1, typename v2>
01682             using div_t = typename div<v1, v2>::type;
01684             template<typename v1, typename v2>
01685             using eq_t = typename eq<v1, v2>::type;
01687             template<typename v1, typename v2>
01688             using gt_t = typename gt<v1, v2>::type;
01690             template<typename v1>
01691             using pos_t = typename pos<v1>::type;
01692
01693             template<typename v>
01694             static constexpr bool pos_v = pos_t<v>::value;
01695         };
01696
01697         template<typename Ring, typename E = void>
01698         requires IsEuclideanDomain<Ring>
01699         struct FractionFieldImpl {};
01700
01701         // fraction field of a field is the field itself
01702         template<typename Field>
01703         requires IsEuclideanDomain<Field>
01704         struct FractionFieldImpl<Field, std::enable_if_t<Field::is_field» {
01705             using type = Field;
01706             template<typename v>
01707             using inject_t = v;
01708         };
01709
01710         // fraction field of a ring is the actual fraction field
01711         template<typename Ring>
01712         requires IsEuclideanDomain<Ring>
01713         struct FractionFieldImpl<Ring, std::enable_if_t<!Ring::is_field» {
01714             using type = _FractionField<Ring>;
01715         };
01716     }
01717
01718     template<typename Ring>
01719     requires IsEuclideanDomain<Ring>
01720     using FractionField = typename internal::FractionFieldImpl<Ring>::type;
01721 }
01722
01723 // short names for common types
01724 namespace aerobus {
01726     using q32 = FractionField<i32>;
01728     using fpq32 = FractionField<polynomial<q32»;
01730     using q64 = FractionField<i64>;
01732     using pi64 = polynomial<i64>;
01734     using fpq64 = FractionField<polynomial<q64»;
01739     template<typename Ring, typename v1, typename v2>
01740     using makefraction_t = typename FractionField<Ring>::template val<v1, v2>;
01741
01742     template<typename Ring, typename v1, typename v2>
01743     using addfractions_t = typename FractionField<Ring>::template add_t<v1, v2>;
```

```
01744     template<typename Ring, typename v1, typename v2>
01745     using mulfractions_t = typename FractionField<Ring>::template mul_t<v1, v2>;
01746 }
01747
01748 // taylor series and common integers (factorial, bernouilli...) appearing in taylor coefficients
01749 namespace aerobus {
01750     namespace internal {
01751         template<typename T, size_t x, typename E = void>
01752         struct factorial {};
01753
01754         template<typename T, size_t x>
01755         struct factorial<T, x, std::enable_if_t<(x > 0)>> {
01756         private:
01757             template<typename, size_t, typename>
01758             friend struct factorial;
01759         public:
01760             using type = typename T::template mul_t<typename T::template val<x>, typename factorial<T,
      x - 1>::type>;
01761             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01762         };
01763
01764         template<typename T>
01765         struct factorial<T, 0> {
01766         public:
01767             using type = typename T::one;
01768             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01769         };
01770     }
01771
01775     template<typename T, size_t i>
01776     using factorial_t = typename internal::factorial<T, i>::type;
01777
01778     template<typename T, size_t i>
01779     inline constexpr typename T::inner_type factorial_v = internal::factorial<T, i>::value;
01780
01781     namespace internal {
01782         template<typename T, size_t k, size_t n, typename E = void>
01783         struct combination_helper {};
01784
01785         template<typename T, size_t k, size_t n>
01786         struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k <= (n / 2) && k > 0)>> {
01787             using type = typename FractionField<T>::template mul_t<
01788                 typename combination_helper<T, k - 1, n - 1>::type,
01789                 makefraction_t<T, typename T::template val<n>, typename T::template val<k>>;
01790         };
01791
01792         template<typename T, size_t k, size_t n>
01793         struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k > (n / 2) && k > 0)>> {
01794             using type = typename combination_helper<T, n - k, n>::type;
01795         };
01796
01797         template<typename T, size_t n>
01798         struct combination_helper<T, 0, n> {
01799             using type = typename FractionField<T>::one;
01800         };
01801
01802         template<typename T, size_t k, size_t n>
01803         struct combination {
01804             using type = typename internal::combination_helper<T, k, n>::type::x;
01805             static constexpr typename T::inner_type value = internal::combination_helper<T, k,
      n>::type::template get<typename T::inner_type>();
01806         };
01807     }
01808
01811     template<typename T, size_t k, size_t n>
01812     using combination_t = typename internal::combination<T, k, n>::type;
01813
01814     template<typename T, size_t k, size_t n>
01815     inline constexpr typename T::inner_type combination_v = internal::combination<T, k, n>::value;
01816
01817     namespace internal {
01818         template<typename T, size_t m>
01819         struct bernouilli;
01820
01821         template<typename T, typename accum, size_t k, size_t m>
01822         struct bernouilli_helper {
01823             using type = typename bernouilli_helper<
01824                 T,
01825                 addfractions_t<T,
01826                     accum,
01827                     mulfractions_t<T,
01828                         makefraction_t<T,
01829                             combination_t<T, k, m + 1>,
01830                             typename T::one>,
01831                         typename bernouilli<T, k>::type
```

```
01832                     >
01833                 >,
01834                 k + 1,
01835                 m>::type;
01836         };
01837
01838         template<typename T, typename accum, size_t m>
01839         struct bernouilli_helper<T, accum, m, m>
01840         {
01841             using type = accum;
01842         };
01843
01844
01845
01846         template<typename T, size_t m>
01847         struct bernouilli {
01848             using type = typename FractionField<T>::template mul_t<
01849                 typename internal::bernouilli_helper<T, typename FractionField<T>::zero, 0, m>::type,
01850                 makefraction_t<T,
01851                 typename T::template val<static_cast<typename T::inner_type>(-1)>,
01852                 typename T::template val<static_cast<typename T::inner_type>(m + 1)>
01853                 >
01854             >;
01855
01856             template<typename floatType>
01857             static constexpr floatType value = type::template get<floatType>();
01858         };
01859
01860         template<typename T>
01861         struct bernouilli<T, 0> {
01862             using type = typename FractionField<T>::one;
01863
01864             template<typename floatType>
01865             static constexpr floatType value = type::template get<floatType>();
01866         };
01867     }
01868
01872     template<typename T, size_t n>
01873     using bernouilli_t = typename internal::bernouilli<T, n>::type;
01874
01875     template<typename FloatType, typename T, size_t n >
01876     inline constexpr FloatType bernouilli_v = internal::bernouilli<T, n>::template value<FloatType>;
01877
01878     namespace internal {
01879         template<typename T, int k, typename E = void>
01880         struct alternate {};
01881
01882         template<typename T, int k>
01883         struct alternate<T, k, std::enable_if_t<k % 2 == 0» {
01884             using type = typename T::one;
01885             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01886         };
01887
01888         template<typename T, int k>
01889         struct alternate<T, k, std::enable_if_t<k % 2 != 0» {
01890             using type = typename T::template sub_t<typename T::zero, typename T::one>;
01891             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01892         };
01893     }
01894
01897     template<typename T, int k>
01898     using alternate_t = typename internal::alternate<T, k>::type;
01899
01900     template<typename T, size_t k>
01901     inline constexpr typename T::inner_type alternate_v = internal::alternate<T, k>::value;
01902
01903     // pow
01904     namespace internal {
01905         template<typename T, auto p, auto n>
01906         struct pow {
01907             using type = typename T::template mul_t<typename T::template val<p>, typename pow<T, p, n
      - 1>::type>;
01908         };
01909
01910         template<typename T, auto p>
01911         struct pow<T, p, 0> { using type = typename T::one; };
01912     }
01913
01914     template<typename T, auto p, auto n>
01915     using pow_t = typename internal::pow<T, p, n>::type;
01916
01917     namespace internal {
01918         template<typename, template<typename, size_t> typename, class>
01919         struct make_taylor_impl;
01920
```

```
01921         template<typename T, template<typename, size_t> typename coeff_at, size_t... Is>
01922         struct make_taylor_impl<T, coeff_at, std::integer_sequence<size_t, Is...>> {
01923             using type = typename polynomial<FractionField<T>>::template val<typename coeff_at<T,
      Is>::type...>;
01924         };
01925     }
01926
01927     // generic taylor serie, depending on coefficients
01928     template<typename T, template<typename, size_t index> typename coeff_at, size_t deg>
01929     using taylor = typename internal::make_taylor_impl<T, coeff_at,
      internal::make_index_sequence_reverse<deg + 1>>::type;
01930
01931     namespace internal {
01932         template<typename T, size_t i>
01933         struct exp_coeff {
01934             using type = makefraction_t<T, typename T::one, factorial_t<T, i>;
01935         };
01936
01937         template<typename T, size_t i, typename E = void>
01938         struct sin_coeff_helper {};
01939
01940         template<typename T, size_t i>
01941         struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0>> {
01942             using type = typename FractionField<T>::zero;
01943         };
01944
01945         template<typename T, size_t i>
01946         struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1>> {
01947             using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i>;
01948         };
01949
01950         template<typename T, size_t i>
01951         struct sin_coeff {
01952             using type = typename sin_coeff_helper<T, i>::type;
01953         };
01954
01955         template<typename T, size_t i, typename E = void>
01956         struct sh_coeff_helper {};
01957
01958         template<typename T, size_t i>
01959         struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0>> {
01960             using type = typename FractionField<T>::zero;
01961         };
01962
01963         template<typename T, size_t i>
01964         struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1>> {
01965             using type = makefraction_t<T, typename T::one, factorial_t<T, i>;
01966         };
01967
01968         template<typename T, size_t i>
01969         struct sh_coeff {
01970             using type = typename sh_coeff_helper<T, i>::type;
01971         };
01972
01973         template<typename T, size_t i, typename E = void>
01974         struct cos_coeff_helper {};
01975
01976         template<typename T, size_t i>
01977         struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1>> {
01978             using type = typename FractionField<T>::zero;
01979         };
01980
01981         template<typename T, size_t i>
01982         struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0>> {
01983             using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i>;
01984         };
01985
01986         template<typename T, size_t i>
01987         struct cos_coeff {
01988             using type = typename cos_coeff_helper<T, i>::type;
01989         };
01990
01991         template<typename T, size_t i, typename E = void>
01992         struct cosh_coeff_helper {};
01993
01994         template<typename T, size_t i>
01995         struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1>> {
01996             using type = typename FractionField<T>::zero;
01997         };
01998
01999         template<typename T, size_t i>
02000         struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0>> {
02001             using type = makefraction_t<T, typename T::one, factorial_t<T, i>;
02002         };
02003
02004         template<typename T, size_t i>
02005         struct cosh_coeff {
```

```
02006             using type = typename cosh_coeff_helper<T, i>::type;
02007         };
02008
02009         template<typename T, size_t i>
02010         struct geom_coeff { using type = typename FractionField<T>::one; };
02011
02012
02013         template<typename T, size_t i, typename E = void>
02014         struct atan_coeff_helper;
02015
02016         template<typename T, size_t i>
02017         struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
02018             using type = makefraction_t<T, alternate_t<T, i / 2>, typename T::template val<i»;
02019         };
02020
02021         template<typename T, size_t i>
02022         struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
02023             using type = typename FractionField<T>::zero;
02024         };
02025
02026         template<typename T, size_t i>
02027         struct atan_coeff { using type = typename atan_coeff_helper<T, i>::type; };
02028
02029         template<typename T, size_t i, typename E = void>
02030         struct asin_coeff_helper;
02031
02032         template<typename T, size_t i>
02033         struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
02034         {
02035             using type = makefraction_t<T,
02036                 factorial_t<T, i - 1>,
02037                 typename T::template mul_t<
02038                     typename T::template val<i>,
02039                     T::template mul_t<
02040                         pow_t<T, 4, i / 2>,
02041                         pow<T, factorial<T, i / 2>::value, 2
02042                     >
02043                 >
02044                 »;
02045         };
02046
02047         template<typename T, size_t i>
02048         struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
02049         {
02050             using type = typename FractionField<T>::zero;
02051         };
02052
02053         template<typename T, size_t i>
02054         struct asin_coeff {
02055             using type = typename asin_coeff_helper<T, i>::type;
02056         };
02057
02058         template<typename T, size_t i>
02059         struct lnp1_coeff {
02060             using type = makefraction_t<T,
02061                 alternate_t<T, i + 1>,
02062                 typename T::template val<i»;
02063         };
02064
02065         template<typename T>
02066         struct lnp1_coeff<T, 0> { using type = typename FractionField<T>::zero; };
02067
02068         template<typename T, size_t i, typename E = void>
02069         struct asinh_coeff_helper;
02070
02071         template<typename T, size_t i>
02072         struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
02073         {
02074             using type = makefraction_t<T,
02075                 typename T::template mul_t<
02076                     alternate_t<T, i / 2>,
02077                     factorial_t<T, i - 1>
02078                 >,
02079                 typename T::template mul_t<
02080                     T::template mul_t<
02081                         typename T::template val<i>,
02082                         pow_t<T, (factorial<T, i / 2>::value), 2>
02083                     >,
02084                     pow_t<T, 4, i / 2>
02085                 >
02086             >;
02087         };
02088
02089         template<typename T, size_t i>
02090         struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
02091         {
02092             using type = typename FractionField<T>::zero;
```

```
02093         };
02094
02095         template<typename T, size_t i>
02096         struct asinh_coeff {
02097             using type = typename asinh_coeff_helper<T, i>::type;
02098         };
02099
02100         template<typename T, size_t i, typename E = void>
02101         struct atanh_coeff_helper;
02102
02103         template<typename T, size_t i>
02104         struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1»
02105         {
02106             // 1/i
02107             using type = typename FractionField<T>:: template val<
02108                 typename T::one,
02109                 typename T::template val<static_cast<typename T::inner_type>(i)»;
02110         };
02111
02112         template<typename T, size_t i>
02113         struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0»
02114         {
02115             using type = typename FractionField<T>::zero;
02116         };
02117
02118         template<typename T, size_t i>
02119         struct atanh_coeff {
02120             using type = typename asinh_coeff_helper<T, i>::type;
02121         };
02122
02123         template<typename T, size_t i, typename E = void>
02124         struct tan_coeff_helper;
02125
02126         template<typename T, size_t i>
02127         struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
02128             using type = typename FractionField<T>::zero;
02129         };
02130
02131         template<typename T, size_t i>
02132         struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
02133         private:
02134             // 4^((i+1)/2)
02135             using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
02136             // 4^((i+1)/2) - 1
02137             using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename
      FractionField<T>::one>;
02138             // (-1)^((i-1)/2)
02139             using altp = typename FractionField<T>::template inject_t<alternate_t<T, (i - 1) / 2»;
02140             using dividend = typename FractionField<T>::template mul_t<
02141                 altp,
02142                 FractionField<T>::template mul_t<
02143                 _4p,
02144                 FractionField<T>::template mul_t<
02145                 _4pm1,
02146                 bernouilli_t<T, (i + 1)>
02147                 >
02148                 >
02149             >;
02150         public:
02151             using type = typename FractionField<T>::template div_t<dividend,
02152                 typename FractionField<T>::template inject_t<factorial_t<T, i + 1»>;
02153         };
02154
02155         template<typename T, size_t i>
02156         struct tan_coeff {
02157             using type = typename tan_coeff_helper<T, i>::type;
02158         };
02159
02160         template<typename T, size_t i, typename E = void>
02161         struct tanh_coeff_helper;
02162
02163         template<typename T, size_t i>
02164         struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
02165             using type = typename FractionField<T>::zero;
02166         };
02167
02168         template<typename T, size_t i>
02169         struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
02170         private:
02171             using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
02172             using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename
      FractionField<T>::one>;
02173             using dividend =
02174                 typename FractionField<T>::template mul_t<
02175                 _4p,
02176                 typename FractionField<T>::template mul_t<
02177                 _4pm1,
```

```
02178                 bernouilli_t<T, (i + 1)>
02179                 >
02180                 >::type;
02181         public:
02182             using type = typename FractionField<T>::template div_t<dividend,
02183                 FractionField<T>::template inject_t<factorial_t<T, i + 1>>>;
02184         };
02185 
02186         template<typename T, size_t i>
02187         struct tanh_coeff {
02188             using type = typename tanh_coeff_helper<T, i>::type;
02189         };
02190     }
02191 
02195     template<typename T, size_t deg>
02196     using exp = taylor<T, internal::exp_coeff, deg>;
02197 
02201     template<typename T, size_t deg>
02202     using expm1 = typename polynomial<FractionField<T>>::template sub_t<
02203         exp<T, deg>,
02204         typename polynomial<FractionField<T>>::one>;
02205 
02209     template<typename T, size_t deg>
02210     using lnp1 = taylor<T, internal::lnp1_coeff, deg>;
02211 
02215     template<typename T, size_t deg>
02216     using atan = taylor<T, internal::atan_coeff, deg>;
02217 
02221     template<typename T, size_t deg>
02222     using sin = taylor<T, internal::sin_coeff, deg>;
02223 
02227     template<typename T, size_t deg>
02228     using sinh = taylor<T, internal::sh_coeff, deg>;
02229 
02233     template<typename T, size_t deg>
02234     using cosh = taylor<T, internal::cosh_coeff, deg>;
02235 
02239     template<typename T, size_t deg>
02240     using cos = taylor<T, internal::cos_coeff, deg>;
02241 
02245     template<typename T, size_t deg>
02246     using geometric_sum = taylor<T, internal::geom_coeff, deg>;
02247 
02251     template<typename T, size_t deg>
02252     using asin = taylor<T, internal::asin_coeff, deg>;
02253 
02257     template<typename T, size_t deg>
02258     using asinh = taylor<T, internal::asinh_coeff, deg>;
02259 
02263     template<typename T, size_t deg>
02264     using atanh = taylor<T, internal::atanh_coeff, deg>;
02265 
02269     template<typename T, size_t deg>
02270     using tan = taylor<T, internal::tan_coeff, deg>;
02271 
02275     template<typename T, size_t deg>
02276     using tanh = taylor<T, internal::tanh_coeff, deg>;
02277 }
02278 
02279 // continued fractions
02280 namespace aerobus {
02283     template<int64_t... values>
02284     struct ContinuedFraction {};
02285 
02286     template<int64_t a0>
02287     struct ContinuedFraction<a0> {
02288         using type = typename q64::template inject_constant_t<a0>;
02289         static constexpr double val = type::template get<double>();
02290     };
02291 
02292     template<int64_t a0, int64_t... rest>
02293     struct ContinuedFraction<a0, rest...> {
02294         using type = q64::template add_t<
02295                 typename q64::template inject_constant_t<a0>,
02296                 typename q64::template div_t<
02297                     typename q64::one,
02298                     typename ContinuedFraction<rest...>::type
02299                 »;
02300         static constexpr double val = type::template get<double>();
02301     };
02302 
02307     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>;
02310     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>;
02312     using SQRT2_fraction =
      ContinuedFraction<1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2>;
```

```
02314     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>;
02315 }
02316
02317 // known polynomials
02318 namespace aerobus {
02319     namespace internal {
02320         template<int kind, int deg>
02321         struct chebyshev_helper {
02322             using type = typename pi64::template sub_t<
02323                 typename pi64::template mul_t<
02324                     typename pi64::template mul_t<
02325                         pi64::inject_constant_t<2>,
02326                         typename pi64::X
02327                     >,
02328                     typename chebyshev_helper<kind, deg-1>::type
02329                 >,
02330                 typename chebyshev_helper<kind, deg-2>::type
02331             >;
02332         };
02333
02334         template<>
02335         struct chebyshev_helper<1, 0> {
02336             using type = typename pi64::one;
02337         };
02338
02339         template<>
02340         struct chebyshev_helper<1, 1> {
02341             using type = typename pi64::X;
02342         };
02343
02344         template<>
02345         struct chebyshev_helper<2, 0> {
02346             using type = typename pi64::one;
02347         };
02348
02349         template<>
02350         struct chebyshev_helper<2, 1> {
02351             using type = typename pi64::template mul_t<
02352                             typename pi64::inject_constant_t<2>,
02353                             typename pi64::X>;
02354         };
02355     }
02356
02359     template<size_t deg>
02360     using chebyshev_T = typename internal::chebyshev_helper<1, deg>::type;
02361
02364     template<size_t deg>
02365     using chebyshev_U = typename internal::chebyshev_helper<2, deg>::type;
02366 }
```

Chapter 7

Examples

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