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 #ifndef __INC_AEROBUS__ // NOLINT
00003 #define __INC_AEROBUS__
00004
00005 #include <cstdint>
00006 #include <cstddef>
00007 #include <cstring>
00008 #include <type_traits>
00009 #include <utility>
00010 #include <algorithm>
00011 #include <functional>
00012 #include <string>
00013 #include <concepts> // NOLINT
00014 #include <array>
00015
00016
00017 #ifdef _MSC_VER
00018 #define ALIGNED(x) __declspec(align(x))
00019 #define INLINED __forceinline
00020 #else
00021 #define ALIGNED(x) __attribute__((aligned(x)))
00022 #define INLINED __attribute__((always_inline)) inline
00023 #endif
00024
00025 // aligned allocation
00026 namespace aerobus {
00033     template<typename T>
00034     T* aligned_malloc(size_t count, size_t alignment) {
00035         #ifdef _MSC_VER
00036         return static_cast<T*>(_aligned_malloc(count * sizeof(T), alignment));
00037         #else
00038         return static_cast<T*>(aligned_alloc(alignment, count * sizeof(T)));
00039         #endif
00040     }
00041
00042     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 } }; // NOLINT
00043
00052     template<typename T, size_t N>
00053     constexpr bool contains(const std::array<T, N>& arr, const T& v) {
00054         for (const auto& vv : arr) {
00055             if (v == vv) {
00056                 return true;
00057             }
00058         }
00059
00060         return false;
00061     }
00062
00063 }  // namespace aerobus
00064
00065 // concepts
00066 namespace aerobus {
00068     template <typename R>
00069     concept IsRing = requires {
00070         typename R::one;
00071         typename R::zero;
00072         typename R::template add_t<typename R::one, typename R::one>;
00073         typename R::template sub_t<typename R::one, typename R::one>;
00074         typename R::template mul_t<typename R::one, typename R::one>;
00075     };
00076
00078     template <typename R>
00079     concept IsEuclideanDomain = IsRing<R> && requires {
00080         typename R::template div_t<typename R::one, typename R::one>;
00081         typename R::template mod_t<typename R::one, typename R::one>;
00082         typename R::template gcd_t<typename R::one, typename R::one>;
00083         typename R::template eq_t<typename R::one, typename R::one>;
00084         typename R::template pos_t<typename R::one>;
00085
00086         R::template pos_v<typename R::one> == true;
00087         // typename R::template gt_t<typename R::one, typename R::zero>;
00088         R::is_euclidean_domain == true;
00089     };
00090
00092     template<typename R>
00093     concept IsField = IsEuclideanDomain<R> && requires {
00094         R::is_field == true;
00095     };
00096 }  // namespace aerobus
00097
00098 // utilities
00099 namespace aerobus {
00100     namespace internal {
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             static_assert(i < sizeof...(Ts) + 1, "index out of range");
00113             using type = typename type_at<i - 1, Ts...>::type;
00114         };
00115
00116         template <typename T, typename... Ts> struct type_at<0, T, Ts...> {
00117             using type = T;
00118         };
00119
00120         template <size_t i, typename... Ts>
00121         using type_at_t = typename type_at<i, Ts...>::type;
00122
00123
00124         template<int32_t n, int32_t i, typename E = void>
00125         struct _is_prime {};
00126
00127         // first 1000 primes are precomputed and stored in a table
00128         template<int32_t n, int32_t i>
00129         struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (contains<int32_t, 1000>(primes, n))» :
      std::true_type {}; // NOLINT
00130
00131         // first 1000 primes are precomputed and stored in a table
00132         template<int32_t n, int32_t i>
00133         struct _is_prime<n, i, std::enable_if_t<(n < 7920) && (!contains<int32_t, 1000>(primes, n))» :
      std::false_type {}; // NOLINT
00134
00135         template<int32_t n, int32_t i>
00136         struct _is_prime<n, i, std::enable_if_t<
00137             (n >= 7920) &&
00138             (i >= 5 && i * i <= n) &&
00139             (n % i == 0 || n % (i + 2) == 0)» : std::false_type {};
00140
00141
00142         template<int32_t n, int32_t i>
00143         struct _is_prime<n, i, std::enable_if_t<
00144             (n >= 7920) &&
00145             (i >= 5 && i * i <= n) &&
00146             (n % i != 0 && n % (i + 2) != 0)» {
00147             static constexpr bool value = _is_prime<n, i + 6>::value;
00148         };
00149
00150         template<int32_t n, int32_t i>
00151         struct _is_prime<n, i, std::enable_if_t<
00152             (n >= 7920) &&
00153             (i >= 5 && i * i > n)» : std::true_type {};
00154     }  // namespace internal
00155
00158     template<int32_t n>
00159     struct is_prime {
00161         static constexpr bool value = internal::_is_prime<n, 5>::value;
00162     };
00163
00164     namespace internal {
00165         template <std::size_t... Is>
00166         constexpr auto index_sequence_reverse(std::index_sequence<Is...> const&)
00167             -> decltype(std::index_sequence<sizeof...(Is) - 1U - Is...>{});
00168
00169         template <std::size_t N>
00170         using make_index_sequence_reverse
00171             = decltype(index_sequence_reverse(std::make_index_sequence<N>{}));
00172
00178         template<typename Ring, typename E = void>
00179         struct gcd;
00180
00181         template<typename Ring>
00182         struct gcd<Ring, std::enable_if_t<Ring::is_euclidean_domain» {
00183             template<typename A, typename B, typename E = void>
00184             struct gcd_helper {};
00185
00186             // B = 0, A > 0
00187             template<typename A, typename B>
00188             struct gcd_helper<A, B, std::enable_if_t<
00189                 ((B::is_zero_t::value) &&
00190                     (Ring::template gt_t<A, typename Ring::zero>::value))» {
00191                 using type = A;
00192             };
00193
00194             // B = 0, A < 0
00195             template<typename A, typename B>
00196             struct gcd_helper<A, B, std::enable_if_t<
00197                 ((B::is_zero_t::value) &&
00198                     !(Ring::template gt_t<A, typename Ring::zero>::value))» {
```

```
00199                 using type = typename Ring::template sub_t<typename Ring::zero, A>;
00200             };
00201
00202             // B != 0
00203             template<typename A, typename B>
00204             struct gcd_helper<A, B, std::enable_if_t<
00205                 (!B::is_zero_t::value)
00206                 » {
00207             private: // NOLINT
00208                 // A / B
00209                 using k = typename Ring::template div_t<A, B>;
00210                 // A - (A/B)*B = A % B
00211                 using m = typename Ring::template sub_t<A, typename Ring::template mul_t<k, B»;
00212
00213             public:
00214                 using type = typename gcd_helper<B, m>::type;
00215             };
00216
00217             template<typename A, typename B>
00218             using type = typename gcd_helper<A, B>::type;
00219         };
00220     }  // namespace internal
00221
00224     template<typename T, typename A, typename B>
00225     using gcd_t = typename internal::gcd<T>::template type<A, B>;
00226 }  // namespace aerobus
00227
00228 // quotient ring by the principal ideal generated by X
00229 namespace aerobus {
00230     template<typename Ring, typename X>
00231     requires IsRing<Ring>
00232     struct Quotient {
00233         template <typename V>
00234         struct val {
00235          private: // NOLINT
00236             using tmp = typename Ring::template mod_t<V, X>;
00237
00238          public:
00239             using type = std::conditional_t<
00240                 Ring::template pos_v<tmp>,
00241                 tmp,
00242                 typename Ring::template sub_t<typename Ring::zero, tmp>
00243             >;
00244         };
00245
00246         using zero = val<typename Ring::zero>;
00247         using one = val<typename Ring::one>;
00248
00249         template<typename v1, typename v2>
00250         using add_t = val<typename Ring::template add_t<typename v1::type, typename v2::type>>;
00251         template<typename v1, typename v2>
00252         using mul_t = val<typename Ring::template mul_t<typename v1::type, typename v2::type>>;
00253         template<typename v1, typename v2>
00254         using div_t = val<typename Ring::template div_t<typename v1::type, typename v2::type>>;
00255         template<typename v1, typename v2>
00256         using mod_t = val<typename Ring::template mod_t<typename v1::type, typename v2::type>>;
00257         template<typename v1, typename v2>
00258         using eq_t = typename Ring::template eq_t<typename v1::type, typename v2::type>;
00259         template<typename v1>
00260         using pos_t = std::true_type;
00261
00262         template<typename v>
00263         static constexpr bool pos_v = pos_t<v>::value;
00264
00265         static constexpr bool is_euclidean_domain = true;
00266
00267         template<auto x>
00268         using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;
00269
00270         template<typename v>
00271         using inject_ring_t = val<v>;
00272     };
00273 }  // namespace aerobus
00274
00275 // type_list
00276 namespace aerobus {
00278     template <typename... Ts>
00279     struct type_list;
00280
00281     namespace internal {
00282         template <typename T, typename... Us>
00283         struct pop_front_h {
00284             using tail = type_list<Us...>;
00285             using head = T;
00286         };
00287
00288         template <uint64_t index, typename L1, typename L2>
```

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

```
00376     struct type_list<> {
00377         static constexpr size_t length = 0;
00378 
00379         template <typename T>
00380         using push_front = type_list<T>;
00381 
00382         template <typename T>
00383         using push_back = type_list<T>;
00384 
00385         template <typename U>
00386         using concat = U;
00387 
00388         // TODO(jewave): assert index == 0
00389         template <uint64_t index, typename T>
00390         using insert = type_list<T>;
00391     };
00392 }  // namespace aerobus
00393 
00394 // i32
00395 namespace aerobus {
00397     struct i32 {
00398         using inner_type = int32_t;
00401         template<int32_t x>
00402         struct val {
00403             static constexpr int32_t v = x;
00404 
00407             template<typename valueType>
00408             static constexpr valueType get() { return static_cast<valueType>(x); }
00409 
00411             using is_zero_t = std::bool_constant<x == 0>;
00412 
00414             static std::string to_string() {
00415                 return std::to_string(x);
00416             }
00417 
00420             template<typename valueRing>
00421             static constexpr valueRing eval(const valueRing& v) {
00422                 return static_cast<valueRing>(x);
00423             }
00424         };
00425 
00427         using zero = val<0>;
00429         using one = val<1>;
00431         static constexpr bool is_field = false;
00433         static constexpr bool is_euclidean_domain = true;
00437         template<auto x>
00438         using inject_constant_t = val<static_cast<int32_t>(x)>;
00439 
00440         template<typename v>
00441         using inject_ring_t = v;
00442 
00443      private:
00444         template<typename v1, typename v2>
00445         struct add {
00446             using type = val<v1::v + v2::v>;
00447         };
00448 
00449         template<typename v1, typename v2>
00450         struct sub {
00451             using type = val<v1::v - v2::v>;
00452         };
00453 
00454         template<typename v1, typename v2>
00455         struct mul {
00456             using type = val<v1::v* v2::v>;
00457         };
00458 
00459         template<typename v1, typename v2>
00460         struct div {
00461             using type = val<v1::v / v2::v>;
00462         };
00463 
00464         template<typename v1, typename v2>
00465         struct remainder {
00466             using type = val<v1::v % v2::v>;
00467         };
00468 
00469         template<typename v1, typename v2>
00470         struct gt {
00471             using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
00472         };
00473 
00474         template<typename v1, typename v2>
00475         struct lt {
00476             using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
00477         };
00478 
```

```
00479         template<typename v1, typename v2>
00480         struct eq {
00481             using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
00482         };
00483 
00484         template<typename v1>
00485         struct pos {
00486             using type = std::bool_constant<(v1::v > 0)>;
00487         };
00488 
00489     public:
00491         template<typename v1, typename v2>
00492         using add_t = typename add<v1, v2>::type;
00493 
00495         template<typename v1, typename v2>
00496         using sub_t = typename sub<v1, v2>::type;
00497 
00499         template<typename v1, typename v2>
00500         using mul_t = typename mul<v1, v2>::type;
00501 
00503         template<typename v1, typename v2>
00504         using div_t = typename div<v1, v2>::type;
00505 
00507         template<typename v1, typename v2>
00508         using mod_t = typename remainder<v1, v2>::type;
00509 
00511         template<typename v1, typename v2>
00512         using gt_t = typename gt<v1, v2>::type;
00513 
00515         template<typename v1, typename v2>
00516         using lt_t = typename lt<v1, v2>::type;
00517 
00519         template<typename v1, typename v2>
00520         using eq_t = typename eq<v1, v2>::type;
00521 
00523         template<typename v1, typename v2>
00524         using gcd_t = gcd_t<i32, v1, v2>;
00525 
00527         template<typename v>
00528         using pos_t = typename pos<v>::type;
00529 
00530         template<typename v>
00531         static constexpr bool pos_v = pos_t<v>::value;
00532     };
00533 }  // namespace aerobus
00534 
00535 // i64
00536 namespace aerobus {
00538     struct i64 {
00539         using inner_type = int64_t;
00542         template<int64_t x>
00543         struct val {
00544             static constexpr int64_t v = x;
00545 
00548             template<typename valueType>
00549             static constexpr valueType get() { return static_cast<valueType>(x); }
00550 
00552             using is_zero_t = std::bool_constant<x == 0>;
00553 
00555             static std::string to_string() {
00556                 return std::to_string(x);
00557             }
00558 
00561             template<typename valueRing>
00562             static constexpr valueRing eval(const valueRing& v) {
00563                 return static_cast<valueRing>(x);
00564             }
00565         };
00566 
00570         template<auto x>
00571         using inject_constant_t = val<static_cast<int64_t>(x)>;
00572 
00573         template<typename v>
00574         using inject_ring_t = v;
00575 
00577         using zero = val<0>;
00579         using one = val<1>;
00581         static constexpr bool is_field = false;
00583         static constexpr bool is_euclidean_domain = true;
00584 
00585     private:
00586         template<typename v1, typename v2>
00587         struct add {
00588             using type = val<v1::v + v2::v>;
00589         };
00590 
00591         template<typename v1, typename v2>
```

```
00592         struct sub {
00593             using type = val<v1::v - v2::v>;
00594         };
00595 
00596         template<typename v1, typename v2>
00597         struct mul {
00598             using type = val<v1::v* v2::v>;
00599         };
00600 
00601         template<typename v1, typename v2>
00602         struct div {
00603             using type = val<v1::v / v2::v>;
00604         };
00605 
00606         template<typename v1, typename v2>
00607         struct remainder {
00608             using type = val<v1::v% v2::v>;
00609         };
00610 
00611         template<typename v1, typename v2>
00612         struct gt {
00613             using type = std::conditional_t<(v1::v > v2::v), std::true_type, std::false_type>;
00614         };
00615 
00616         template<typename v1, typename v2>
00617         struct lt {
00618             using type = std::conditional_t<(v1::v < v2::v), std::true_type, std::false_type>;
00619         };
00620 
00621         template<typename v1, typename v2>
00622         struct eq {
00623             using type = std::conditional_t<(v1::v == v2::v), std::true_type, std::false_type>;
00624         };
00625 
00626         template<typename v>
00627         struct pos {
00628             using type = std::bool_constant<(v::v > 0)>;
00629         };
00630 
00631     public:
00633         template<typename v1, typename v2>
00634         using add_t = typename add<v1, v2>::type;
00635 
00637         template<typename v1, typename v2>
00638         using sub_t = typename sub<v1, v2>::type;
00639 
00641         template<typename v1, typename v2>
00642         using mul_t = typename mul<v1, v2>::type;
00643 
00645         template<typename v1, typename v2>
00646         using div_t = typename div<v1, v2>::type;
00647 
00649         template<typename v1, typename v2>
00650         using mod_t = typename remainder<v1, v2>::type;
00651 
00653         template<typename v1, typename v2>
00654         using gt_t = typename gt<v1, v2>::type;
00655 
00657         template<typename v1, typename v2>
00658         using lt_t = typename lt<v1, v2>::type;
00659 
00661         template<typename v1, typename v2>
00662         using eq_t = typename eq<v1, v2>::type;
00663 
00665         template<typename v1, typename v2>
00666         using gcd_t = gcd_t<i64, v1, v2>;
00667 
00669         template<typename v>
00670         using pos_t = typename pos<v>::type;
00671 
00672         template<typename v>
00673         static constexpr bool pos_v = pos_t<v>::value;
00674     };
00675 }  // namespace aerobus
00676 
00677 // z/pz
00678 namespace aerobus {
00683     template<int32_t p>
00684     struct zpz {
00685         using inner_type = int32_t;
00686         template<int32_t x>
00687         struct val {
00688             static constexpr int32_t v = x % p;
00689 
00690             template<typename valueType>
00691             static constexpr valueType get() { return static_cast<valueType>(x % p); }
00692 
```

```
00693             using is_zero_t = std::bool_constant<x% p == 0>;
00694             static std::string to_string() {
00695                 return std::to_string(x % p);
00696             }
00697 
00698             template<typename valueRing>
00699             static constexpr valueRing eval(const valueRing& v) {
00700                 return static_cast<valueRing>(x % p);
00701             }
00702         };
00703 
00704         template<auto x>
00705         using inject_constant_t = val<static_cast<int32_t>(x)>;
00706 
00707         using zero = val<0>;
00708         using one = val<1>;
00709         static constexpr bool is_field = is_prime<p>::value;
00710         static constexpr bool is_euclidean_domain = true;
00711 
00712     private:
00713         template<typename v1, typename v2>
00714         struct add {
00715             using type = val<(v1::v + v2::v) % p>;
00716         };
00717 
00718         template<typename v1, typename v2>
00719         struct sub {
00720             using type = val<(v1::v - v2::v) % p>;
00721         };
00722 
00723         template<typename v1, typename v2>
00724         struct mul {
00725             using type = val<(v1::v* v2::v) % p>;
00726         };
00727 
00728         template<typename v1, typename v2>
00729         struct div {
00730             using type = val<(v1::v% p) / (v2::v % p)>;
00731         };
00732 
00733         template<typename v1, typename v2>
00734         struct remainder {
00735             using type = val<(v1::v% v2::v) % p>;
00736         };
00737 
00738         template<typename v1, typename v2>
00739         struct gt {
00740             using type = std::conditional_t<(v1::v% p > v2::v% p), std::true_type, std::false_type>;
00741         };
00742 
00743         template<typename v1, typename v2>
00744         struct lt {
00745             using type = std::conditional_t<(v1::v% p < v2::v% p), std::true_type, std::false_type>;
00746         };
00747 
00748         template<typename v1, typename v2>
00749         struct eq {
00750             using type = std::conditional_t<(v1::v% p == v2::v % p), std::true_type, std::false_type>;
00751         };
00752 
00753         template<typename v1>
00754         struct pos {
00755             using type = std::bool_constant<(v1::v > 0)>;
00756         };
00757 
00758     public:
00759         template<typename v1, typename v2>
00760         using add_t = typename add<v1, v2>::type;
00761 
00762         template<typename v1, typename v2>
00763         using sub_t = typename sub<v1, v2>::type;
00764 
00765         template<typename v1, typename v2>
00766         using mul_t = typename mul<v1, v2>::type;
00767 
00768         template<typename v1, typename v2>
00769         using div_t = typename div<v1, v2>::type;
00770 
00771         template<typename v1, typename v2>
00772         using mod_t = typename remainder<v1, v2>::type;
00773 
00774         template<typename v1, typename v2>
00775         using gt_t = typename gt<v1, v2>::type;
00776 
00777         template<typename v1, typename v2>
00778         using lt_t = typename lt<v1, v2>::type;
00779 
```

```
00780         template<typename v1, typename v2>
00781         using eq_t = typename eq<v1, v2>::type;
00782 
00783         template<typename v1, typename v2>
00784         using gcd_t = gcd_t<i32, v1, v2>;
00785 
00786         template<typename v1>
00787         using pos_t = typename pos<v1>::type;
00788 
00789         template<typename v>
00790         static constexpr bool pos_v = pos_t<v>::value;
00791     };
00792 }  // namespace aerobus
00793 
00794 // polynomial
00795 namespace aerobus {
00796     // coeffN x^N + ...
00801     template<typename Ring, char variable_name = 'x'>
00802     requires IsEuclideanDomain<Ring>
00803     struct polynomial {
00804         static constexpr bool is_field = false;
00805         static constexpr bool is_euclidean_domain = Ring::is_euclidean_domain;
00806 
00807         template<typename coeffN, typename... coeffs>
00808         struct val {
00810             static constexpr size_t degree = sizeof...(coeffs);
00812             using aN = coeffN;
00814             using strip = val<coeffs...>;
00816             using is_zero_t = std::bool_constant<(degree == 0) && (aN::is_zero_t::value)>;
00817 
00818          private:
00819             template<size_t index, typename E = void>
00820             struct coeff_at {};
00821 
00822             template<size_t index>
00823             struct coeff_at<index, std::enable_if_t<(index >= 0 && index <= sizeof...(coeffs))>> {
00824                 using type = internal::type_at_t<sizeof...(coeffs) - index, coeffN, coeffs...>;
00825             };
00826 
00827             template<size_t index>
00828             struct coeff_at<index, std::enable_if_t<(index < 0 || index > sizeof...(coeffs))>> {
00829                 using type = typename Ring::zero;
00830             };
00831 
00832          public:
00835             template<size_t index>
00836             using coeff_at_t = typename coeff_at<index>::type;
00837 
00840             static std::string to_string() {
00841                 return string_helper<coeffN, coeffs...>::func();
00842             }
00843 
00848             template<typename valueRing>
00849             static constexpr valueRing eval(const valueRing& x) {
00850                 return eval_helper<valueRing, val>::template inner<0, degree +
      1>::func(static_cast<valueRing>(0), x);
00851             }
00852         };
00853 
00854         // specialization for constants
00855         template<typename coeffN>
00856         struct val<coeffN> {
00857             static constexpr size_t degree = 0;
00858             using aN = coeffN;
00859             using strip = val<coeffN>;
00860             using is_zero_t = std::bool_constant<aN::is_zero_t::value>;
00861 
00862             template<size_t index, typename E = void>
00863             struct coeff_at {};
00864 
00865             template<size_t index>
00866             struct coeff_at<index, std::enable_if_t<(index == 0)>> {
00867                 using type = aN;
00868             };
00869 
00870             template<size_t index>
00871             struct coeff_at<index, std::enable_if_t<(index < 0 || index > 0)>> {
00872                 using type = typename Ring::zero;
00873             };
00874 
00875             template<size_t index>
00876             using coeff_at_t = typename coeff_at<index>::type;
00877 
00878             static std::string to_string() {
00879                 return string_helper<coeffN>::func();
00880             }
00881 
```

```
00882             template<typename valueRing>
00883             static constexpr valueRing eval(const valueRing& x) {
00884                 return static_cast<valueRing>(aN::template get<valueRing>());
00885             }
00886         };
00887 
00889         using zero = val<typename Ring::zero>;
00891         using one = val<typename Ring::one>;
00893         using X = val<typename Ring::one, typename Ring::zero>;
00894 
00895     private:
00896         template<typename P, typename E = void>
00897         struct simplify;
00898 
00899         template <typename P1, typename P2, typename I>
00900         struct add_low;
00901 
00902         template<typename P1, typename P2>
00903         struct add {
00904             using type = typename simplify<typename add_low<
00905             P1,
00906             P2,
00907             internal::make_index_sequence_reverse<
00908             std::max(P1::degree, P2::degree) + 1
00909             »::type>::type;
00910         };
00911 
00912         template <typename P1, typename P2, typename I>
00913         struct sub_low;
00914 
00915         template <typename P1, typename P2, typename I>
00916         struct mul_low;
00917 
00918         template<typename v1, typename v2>
00919         struct mul {
00920                 using type = typename mul_low<
00921                     v1,
00922                     v2,
00923                     internal::make_index_sequence_reverse<
00924                     v1::degree + v2::degree + 1
00925                     »::type;
00926         };
00927 
00928         template<typename coeff, size_t deg>
00929         struct monomial;
00930 
00931         template<typename v, typename E = void>
00932         struct derive_helper {};
00933 
00934         template<typename v>
00935         struct derive_helper<v, std::enable_if_t<v::degree == 0» {
00936             using type = zero;
00937         };
00938 
00939         template<typename v>
00940         struct derive_helper<v, std::enable_if_t<v::degree != 0» {
00941             using type = typename add<
00942                 typename derive_helper<typename simplify<typename v::strip>::type>::type,
00943                 typename monomial<
00944                     typename Ring::template mul_t<
00945                         typename v::aN,
00946                         typename Ring::template inject_constant_t<(v::degree)>
00947                     >,
00948                     v::degree - 1
00949                 >::type
00950             >::type;
00951         };
00952 
00953         template<typename v1, typename v2, typename E = void>
00954         struct eq_helper {};
00955 
00956         template<typename v1, typename v2>
00957         struct eq_helper<v1, v2, std::enable_if_t<v1::degree != v2::degree» {
00958             using type = std::false_type;
00959         };
00960 
00961 
00962         template<typename v1, typename v2>
00963         struct eq_helper<v1, v2, std::enable_if_t<
00964             v1::degree == v2::degree &&
00965             (v1::degree != 0 || v2::degree != 0) &&
00966             std::is_same<
00967             typename Ring::template eq_t<typename v1::aN, typename v2::aN>,
00968             std::false_type
00969             >::value
00970         >
00971         > {
```

```
00972             using type = std::false_type;
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::true_type
00982             >::value
00983         >> {
00984             using type = typename eq_helper<typename v1::strip, typename v2::strip>::type;
00985         };
00986
00987         template<typename v1, typename v2>
00988         struct eq_helper<v1, v2, std::enable_if_t<
00989             v1::degree == v2::degree &&
00990             (v1::degree == 0)
00991         >> {
00992             using type = typename Ring::template eq_t<typename v1::aN, typename v2::aN>;
00993         };
00994
00995         template<typename v1, typename v2, typename E = void>
00996         struct lt_helper {};
00997
00998         template<typename v1, typename v2>
00999         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)>> {
01000             using type = std::true_type;
01001         };
01002
01003         template<typename v1, typename v2>
01004         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree == v2::degree)>> {
01005             using type = typename Ring::template lt_t<typename v1::aN, typename v2::aN>;
01006         };
01007
01008         template<typename v1, typename v2>
01009         struct lt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)>> {
01010             using type = std::false_type;
01011         };
01012
01013         template<typename v1, typename v2, typename E = void>
01014         struct gt_helper {};
01015
01016         template<typename v1, typename v2>
01017         struct gt_helper<v1, v2, std::enable_if_t<(v1::degree > v2::degree)>> {
01018             using type = std::true_type;
01019         };
01020
01021         template<typename v1, typename v2>
01022         struct gt_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>
01027         struct gt_helper<v1, v2, std::enable_if_t<(v1::degree < v2::degree)>> {
01028             using type = std::false_type;
01029         };
01030
01031         // when high power is zero : strip
01032         template<typename P>
01033         struct simplify<P, std::enable_if_t<
01034             std::is_same<
01035             typename Ring::zero,
01036             typename P::aN
01037             >::value && (P::degree > 0)
01038         >> {
01039             using type = typename simplify<typename P::strip>::type;
01040         };
01041
01042         // otherwise : do nothing
01043         template<typename P>
01044         struct simplify<P, std::enable_if_t<
01045             !std::is_same<
01046             typename Ring::zero,
01047             typename P::aN
01048             >::value && (P::degree > 0)
01049         >> {
01050             using type = P;
01051         };
01052
01053         // do not simplify constants
01054         template<typename P>
01055         struct simplify<P, std::enable_if_t<P::degree == 0>> {
01056             using type = P;
01057         };
01058
```

```
01059         // addition at
01060         template<typename P1, typename P2, size_t index>
01061         struct add_at {
01062             using type =
01063                 typename Ring::template add_t<
01064                     typename P1::template coeff_at_t<index>,
01065                     typename P2::template coeff_at_t<index>>;
01066         };
01067
01068         template<typename P1, typename P2, size_t index>
01069         using add_at_t = typename add_at<P1, P2, index>::type;
01070
01071         template<typename P1, typename P2, std::size_t... I>
01072         struct add_low<P1, P2, std::index_sequence<I...>> {
01073             using type = val<add_at_t<P1, P2, I>...>;
01074         };
01075
01076         // substraction at
01077         template<typename P1, typename P2, size_t index>
01078         struct sub_at {
01079             using type =
01080                 typename Ring::template sub_t<
01081                     typename P1::template coeff_at_t<index>,
01082                     typename P2::template coeff_at_t<index>>;
01083         };
01084
01085         template<typename P1, typename P2, size_t index>
01086         using sub_at_t = typename sub_at<P1, P2, index>::type;
01087
01088         template<typename P1, typename P2, std::size_t... I>
01089         struct sub_low<P1, P2, std::index_sequence<I...>> {
01090             using type = val<sub_at_t<P1, P2, I>...>;
01091         };
01092
01093         template<typename P1, typename P2>
01094         struct sub {
01095             using type = typename simplify<typename sub_low<
01096             P1,
01097             P2,
01098             internal::make_index_sequence_reverse<
01099             std::max(P1::degree, P2::degree) + 1
01100             >::type>::type;
01101         };
01102
01103         // multiplication at
01104         template<typename v1, typename v2, size_t k, size_t index, size_t stop>
01105         struct mul_at_loop_helper {
01106             using type = typename Ring::template add_t<
01107                 typename Ring::template mul_t<
01108                 typename v1::template coeff_at_t<index>,
01109                 typename v2::template coeff_at_t<k - index>
01110                 >,
01111                 typename mul_at_loop_helper<v1, v2, k, index + 1, stop>::type
01112             >;
01113         };
01114
01115         template<typename v1, typename v2, size_t k, size_t stop>
01116         struct mul_at_loop_helper<v1, v2, k, stop, stop> {
01117             using type = typename Ring::template mul_t<
01118                 typename v1::template coeff_at_t<stop>,
01119                 typename v2::template coeff_at_t<0>>;
01120         };
01121
01122         template <typename v1, typename v2, size_t k, typename E = void>
01123         struct mul_at {};
01124
01125         template<typename v1, typename v2, size_t k>
01126         struct mul_at<v1, v2, k, std::enable_if_t<(k < 0) || (k > v1::degree + v2::degree)>> {
01127             using type = typename Ring::zero;
01128         };
01129
01130         template<typename v1, typename v2, size_t k>
01131         struct mul_at<v1, v2, k, std::enable_if_t<(k >= 0) && (k <= v1::degree + v2::degree)>> {
01132             using type = typename mul_at_loop_helper<v1, v2, k, 0, k>::type;
01133         };
01134
01135         template<typename P1, typename P2, size_t index>
01136         using mul_at_t = typename mul_at<P1, P2, index>::type;
01137
01138         template<typename P1, typename P2, std::size_t... I>
01139         struct mul_low<P1, P2, std::index_sequence<I...>> {
01140             using type = val<mul_at_t<P1, P2, I>...>;
01141         };
01142
01143         // division helper
01144         template< typename A, typename B, typename Q, typename R, typename E = void>
01145         struct div_helper {};
```

```
01146
01147         template<typename A, typename B, typename Q, typename R>
01148         struct div_helper<A, B, Q, R, std::enable_if_t<
01149             (R::degree < B::degree) ||
01150             (R::degree == 0 && std::is_same<typename R::aN, typename Ring::zero>::value)» {
01151             using q_type = Q;
01152             using mod_type = R;
01153             using gcd_type = B;
01154         };
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          private: // NOLINT
01161             using rN = typename R::aN;
01162             using bN = typename B::aN;
01163             using pT = typename monomial<typename Ring::template div_t<rN, bN>, R::degree -
      B::degree>::type;
01164             using rr = typename sub<R, typename mul<pT, B>::type>::type;
01165             using qq = typename add<Q, pT>::type;
01166
01167          public:
01168             using q_type = typename div_helper<A, B, qq, rr>::q_type;
01169             using mod_type = typename div_helper<A, B, qq, rr>::mod_type;
01170             using gcd_type = rr;
01171         };
01172
01173         template<typename A, typename B>
01174         struct div {
01175             static_assert(Ring::is_euclidean_domain, "cannot divide in that type of Ring");
01176             using q_type = typename div_helper<A, B, zero, A>::q_type;
01177             using m_type = typename div_helper<A, B, zero, A>::mod_type;
01178         };
01179
01180         template<typename P>
01181         struct make_unit {
01182             using type = typename div<P, val<typename P::aN>>::q_type;
01183         };
01184
01185         template<typename coeff, size_t deg>
01186         struct monomial {
01187             using type = typename mul<X, typename monomial<coeff, deg - 1>::type>::type;
01188         };
01189
01190         template<typename coeff>
01191         struct monomial<coeff, 0> {
01192             using type = val<coeff>;
01193         };
01194
01195         template<typename valueRing, typename P>
01196         struct eval_helper {
01197             template<size_t index, size_t stop>
01198             struct inner {
01199                 static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
01200                     constexpr valueRing coeff =
01201                         static_cast<valueRing>(P::template coeff_at_t<P::degree - index>::template
      get<valueRing>());
01202                     return eval_helper<valueRing, P>::template inner<index + 1, stop>::func(x * accum
      + coeff, x);
01203                 }
01204             };
01205
01206             template<size_t stop>
01207             struct inner<stop, stop> {
01208                 static constexpr valueRing func(const valueRing& accum, const valueRing& x) {
01209                     return accum;
01210                 }
01211             };
01212         };
01213
01214         template<typename coeff, typename... coeffs>
01215         struct string_helper {
01216             static std::string func() {
01217                 std::string tail = string_helper<coeffs...>::func();
01218                 std::string result = "";
01219                 if (Ring::template eq_t<coeff, typename Ring::zero>::value) {
01220                     return tail;
01221                 } else if (Ring::template eq_t<coeff, typename Ring::one>::value) {
01222                     if (sizeof...(coeffs) == 1) {
01223                         result += std::string(1, variable_name);
01224                     } else {
01225                         result += std::string(1, variable_name) + "^" +
      std::to_string(sizeof...(coeffs));
01226                     }
01227                 } else {
01228                     if (sizeof...(coeffs) == 1) {
```

```
01229                         result += coeff::to_string() + " " + std::string(1, variable_name);
01230                     } else {
01231                         result += coeff::to_string()
01232                                 + " " + std::string(1, variable_name)
01233                                 + "^" + std::to_string(sizeof...(coeffs));
01234                     }
01235                 }
01236 
01237                 if (!tail.empty()) {
01238                     result += " + " + tail;
01239                 }
01240 
01241                 return result;
01242             }
01243         };
01244 
01245         template<typename coeff>
01246         struct string_helper<coeff> {
01247             static std::string func() {
01248                 if (!std::is_same<coeff, typename Ring::zero>::value) {
01249                     return coeff::to_string();
01250                 } else {
01251                     return "";
01252                 }
01253             }
01254         };
01255 
01256     public:
01259         template<typename P>
01260         using simplify_t = typename simplify<P>::type;
01261 
01265         template<typename v1, typename v2>
01266         using add_t = typename add<v1, v2>::type;
01267 
01271         template<typename v1, typename v2>
01272         using sub_t = typename sub<v1, v2>::type;
01273 
01277         template<typename v1, typename v2>
01278         using mul_t = typename mul<v1, v2>::type;
01279 
01283         template<typename v1, typename v2>
01284         using eq_t = typename eq_helper<v1, v2>::type;
01285 
01289         template<typename v1, typename v2>
01290         using lt_t = typename lt_helper<v1, v2>::type;
01291 
01295         template<typename v1, typename v2>
01296         using gt_t = typename gt_helper<v1, v2>::type;
01297 
01301         template<typename v1, typename v2>
01302         using div_t = typename div<v1, v2>::q_type;
01303 
01307         template<typename v1, typename v2>
01308         using mod_t = typename div_helper<v1, v2, zero, v1>::mod_type;
01309 
01313         template<typename coeff, size_t deg>
01314         using monomial_t = typename monomial<coeff, deg>::type;
01315 
01318         template<typename v>
01319         using derive_t = typename derive_helper<v>::type;
01320 
01323         template<typename v>
01324         using pos_t = typename Ring::template pos_t<typename v::aN>;
01325 
01326         template<typename v>
01327         static constexpr bool pos_v = pos_t<v>::value;
01328 
01332         template<typename v1, typename v2>
01333         using gcd_t = std::conditional_t<
01334             Ring::is_euclidean_domain,
01335             typename make_unit<gcd_t<polynomial<Ring, variable_name>, v1, v2»::type,
01336             void>;
01337 
01341         template<auto x>
01342         using inject_constant_t = val<typename Ring::template inject_constant_t<x>>;
01343 
01347         template<typename v>
01348         using inject_ring_t = val<v>;
01349     };
01350 }  // namespace aerobus
01351 
01352 // fraction field
01353 namespace aerobus {
01354     namespace internal {
01355         template<typename Ring, typename E = void>
01356         requires IsEuclideanDomain<Ring>
01357         struct _FractionField {};
```

```
01358
01359         template<typename Ring>
01360         requires IsEuclideanDomain<Ring>
01361         struct _FractionField<Ring, std::enable_if_t<Ring::is_euclidean_domain» {
01363             static constexpr bool is_field = true;
01364             static constexpr bool is_euclidean_domain = true;
01365
01366          private:
01367             template<typename val1, typename val2, typename E = void>
01368             struct to_string_helper {};
01369
01370             template<typename val1, typename val2>
01371             struct to_string_helper <val1, val2,
01372                 std::enable_if_t<
01373                 Ring::template eq_t<
01374                 val2, typename Ring::one
01375                 >::value
01376                 >
01377             > {
01378                 static std::string func() {
01379                     return val1::to_string();
01380                 }
01381             };
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,
01388                 typename Ring::one
01389                 >::value
01390                 >
01391             > {
01392                 static std::string func() {
01393                     return "(" + val1::to_string() + ") / (" + val2::to_string() + ")";
01394                 }
01395             };
01396
01397          public:
01401             template<typename val1, typename val2>
01402             struct val {
01403                 using x = val1;
01404                 using y = val2;
01406                 using is_zero_t = typename val1::is_zero_t;
01407                 using ring_type = Ring;
01408                 using field_type = _FractionField<Ring>;
01409
01411                  static constexpr bool is_integer = std::is_same<val2, typename Ring::one>::value;
01412
01416                 template<typename valueType>
01417                 static constexpr valueType get() { return static_cast<valueType>(x::v) /
      static_cast<valueType>(y::v); }
01418
01421                 static std::string to_string() {
01422                     return to_string_helper<val1, val2>::func();
01423                 }
01424
01429                 template<typename valueRing>
01430                 static constexpr valueRing eval(const valueRing& v) {
01431                     return x::eval(v) / y::eval(v);
01432                 }
01433             };
01434
01436             using zero = val<typename Ring::zero, typename Ring::one>;
01438             using one = val<typename Ring::one, typename Ring::one>;
01439
01442             template<typename v>
01443             using inject_t = val<v, typename Ring::one>;
01444
01447             template<auto x>
01448             using inject_constant_t = val<typename Ring::template inject_constant_t<x>, typename
      Ring::one>;
01449
01452             template<typename v>
01453             using inject_ring_t = val<typename Ring::template inject_ring_t<v>, typename Ring::one>;
01454
01455             using ring_type = Ring;
01456
01457          private:
01458             template<typename v, typename E = void>
01459             struct simplify {};
01460
01461             // x = 0
01462             template<typename v>
01463             struct simplify<v, std::enable_if_t<v::x::is_zero_t::value» {
01464                 using type = typename _FractionField<Ring>::zero;
01465             };
```

```
01466
01467             // x != 0
01468             template<typename v>
01469             struct simplify<v, std::enable_if_t<!v::x::is_zero_t::value» {
01470              private:
01471                 using _gcd = typename Ring::template gcd_t<typename v::x, typename v::y>;
01472                 using newx = typename Ring::template div_t<typename v::x, _gcd>;
01473                 using newy = typename Ring::template div_t<typename v::y, _gcd>;
01474
01475                 using posx = std::conditional_t<
01476                                 !Ring::template pos_v<newy>,
01477                                 typename Ring::template sub_t<typename Ring::zero, newx>,
01478                                 newx>;
01479                 using posy = std::conditional_t<
01480                                 !Ring::template pos_v<newy>,
01481                                 typename Ring::template sub_t<typename Ring::zero, newy>,
01482                                 newy>;
01483              public:
01484                 using type = typename _FractionField<Ring>::template val<posx, posy>;
01485             };
01486
01487          public:
01490             template<typename v>
01491             using simplify_t = typename simplify<v>::type;
01492
01493          private:
01494             template<typename v1, typename v2>
01495             struct add {
01496              private:
01497                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01498                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01499                 using dividend = typename Ring::template add_t<a, b>;
01500                 using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01501                 using g = typename Ring::template gcd_t<dividend, diviser>;
01502
01503              public:
01504                 using type = typename _FractionField<Ring>::template simplify_t<val<dividend,
      diviser»;
01505             };
01506
01507             template<typename v>
01508             struct pos {
01509                 using type = std::conditional_t<
01510                     (Ring::template pos_v<typename v::x> && Ring::template pos_v<typename v::y>) ||
01511                     (!Ring::template pos_v<typename v::x> && !Ring::template pos_v<typename v::y>),
01512                     std::true_type,
01513                     std::false_type>;
01514             };
01515
01516             template<typename v1, typename v2>
01517             struct sub {
01518              private:
01519                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01520                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01521                 using dividend = typename Ring::template sub_t<a, b>;
01522                 using diviser = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01523                 using g = typename Ring::template gcd_t<dividend, diviser>;
01524
01525              public:
01526                 using type = typename _FractionField<Ring>::template simplify_t<val<dividend,
      diviser»;
01527             };
01528
01529             template<typename v1, typename v2>
01530             struct mul {
01531              private:
01532                 using a = typename Ring::template mul_t<typename v1::x, typename v2::x>;
01533                 using b = typename Ring::template mul_t<typename v1::y, typename v2::y>;
01534
01535              public:
01536                 using type = typename _FractionField<Ring>::template simplify_t<val<a, b»;
01537             };
01538
01539             template<typename v1, typename v2, typename E = void>
01540             struct div {};
01541
01542             template<typename v1, typename v2>
01543             struct div<v1, v2, std::enable_if_t<!std::is_same<v2, typename
      _FractionField<Ring>::zero>::value»  {
01544              private:
01545                 using a = typename Ring::template mul_t<typename v1::x, typename v2::y>;
01546                 using b = typename Ring::template mul_t<typename v1::y, typename v2::x>;
01547
01548              public:
01549                 using type = typename _FractionField<Ring>::template simplify_t<val<a, b»;
01550             };
01551
```

```
01552             template<typename v1, typename v2>
01553             struct div<v1, v2, std::enable_if_t<
01554                 std::is_same<zero, v1>::value && std::is_same<v2, zero>::value»  {
01555                 using type = one;
01556             };
01557
01558             template<typename v1, typename v2>
01559             struct eq {
01560                 using type = std::conditional_t<
01561                         std::is_same<typename simplify_t<v1>::x, typename simplify_t<v2>::x>::value &&
01562                         std::is_same<typename simplify_t<v1>::y, typename simplify_t<v2>::y>::value,
01563                     std::true_type,
01564                     std::false_type>;
01565             };
01566
01567             template<typename TL, typename E = void>
01568             struct vadd {};
01569
01570             template<typename TL>
01571             struct vadd<TL, std::enable_if_t<(TL::length > 1)» {
01572                 using head = typename TL::pop_front::type;
01573                 using tail = typename TL::pop_front::tail;
01574                 using type = typename add<head, typename vadd<tail>::type>::type;
01575             };
01576
01577             template<typename TL>
01578             struct vadd<TL, std::enable_if_t<(TL::length == 1)» {
01579                 using type = typename TL::template at<0>;
01580             };
01581
01582             template<typename... vals>
01583             struct vmul {};
01584
01585             template<typename v1, typename... vals>
01586             struct vmul<v1, vals...> {
01587                 using type = typename mul<v1, typename vmul<vals...>::type>::type;
01588             };
01589
01590             template<typename v1>
01591             struct vmul<v1> {
01592                 using type = v1;
01593             };
01594
01595
01596             template<typename v1, typename v2, typename E = void>
01597             struct gt;
01598
01599             template<typename v1, typename v2>
01600             struct gt<v1, v2, std::enable_if_t<
01601                 (eq<v1, v2>::type::value)
01602                 » {
01603                 using type = std::false_type;
01604             };
01605
01606             template<typename v1, typename v2>
01607             struct gt<v1, v2, std::enable_if_t<
01608                 (!eq<v1, v2>::type::value) &&
01609                 (!pos<v1>::type::value) && (!pos<v2>::type::value)
01610                 » {
01611                 using type = typename gt<
01612                     typename sub<zero, v1>::type, typename sub<zero, v2>::type
01613                 >::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 = std::true_type;
01622             };
01623
01624             template<typename v1, typename v2>
01625             struct gt<v1, v2, std::enable_if_t<
01626                 (!eq<v1, v2>::type::value) &&
01627                 (!pos<v1>::type::value) && (pos<v2>::type::value)
01628                 » {
01629                 using type = std::false_type;
01630             };
01631
01632             template<typename v1, typename v2>
01633             struct gt<v1, v2, std::enable_if_t<
01634                 (!eq<v1, v2>::type::value) &&
01635                 (pos<v1>::type::value) && (pos<v2>::type::value)
01636                 » {
01637                 using type = typename Ring::template gt_t<
01638                     typename Ring::template mul_t<v1::x, v2::y>,
```

```
01639                     typename Ring::template mul_t<v2::y, v2::x>
01640                 >;
01641             };
01642 
01643          public:
01645             template<typename v1, typename v2>
01646             using add_t = typename add<v1, v2>::type;
01648             template<typename v1, typename v2>
01649             using mod_t = zero;
01653             template<typename v1, typename v2>
01654             using gcd_t = v1;
01657             template<typename... vs>
01658             using vadd_t = typename vadd<vs...>::type;
01661             template<typename... vs>
01662             using vmul_t = typename vmul<vs...>::type;
01664             template<typename v1, typename v2>
01665             using sub_t = typename sub<v1, v2>::type;
01667             template<typename v1, typename v2>
01668             using mul_t = typename mul<v1, v2>::type;
01670             template<typename v1, typename v2>
01671             using div_t = typename div<v1, v2>::type;
01673             template<typename v1, typename v2>
01674             using eq_t = typename eq<v1, v2>::type;
01676             template<typename v1, typename v2>
01677             using gt_t = typename gt<v1, v2>::type;
01679             template<typename v1>
01680             using pos_t = typename pos<v1>::type;
01681 
01682             template<typename v>
01683             static constexpr bool pos_v = pos_t<v>::value;
01684         };
01685 
01686         template<typename Ring, typename E = void>
01687         requires IsEuclideanDomain<Ring>
01688         struct FractionFieldImpl {};
01689 
01690         // fraction field of a field is the field itself
01691         template<typename Field>
01692         requires IsEuclideanDomain<Field>
01693         struct FractionFieldImpl<Field, std::enable_if_t<Field::is_field>> {
01694             using type = Field;
01695             template<typename v>
01696             using inject_t = v;
01697         };
01698 
01699         // fraction field of a ring is the actual fraction field
01700         template<typename Ring>
01701         requires IsEuclideanDomain<Ring>
01702         struct FractionFieldImpl<Ring, std::enable_if_t<!Ring::is_field>> {
01703             using type = _FractionField<Ring>;
01704         };
01705     }  // namespace internal
01706 
01707     template<typename Ring>
01708     requires IsEuclideanDomain<Ring>
01709     using FractionField = typename internal::FractionFieldImpl<Ring>::type;
01710 }  // namespace aerobus
01711 
01712 // short names for common types
01713 namespace aerobus {
01715     using q32 = FractionField<i32>;
01717     using fpq32 = FractionField<polynomial<q32>>;
01719     using q64 = FractionField<i64>;
01721     using pi64 = polynomial<i64>;
01723     using fpq64 = FractionField<polynomial<q64>>;
01728     template<typename Ring, typename v1, typename v2>
01729     using makefraction_t = typename FractionField<Ring>::template val<v1, v2>;
01730 
01731     template<typename Ring, typename v1, typename v2>
01732     using addfractions_t = typename FractionField<Ring>::template add_t<v1, v2>;
01733     template<typename Ring, typename v1, typename v2>
01734     using mulfractions_t = typename FractionField<Ring>::template mul_t<v1, v2>;
01735 }  // namespace aerobus
01736 
01737 // taylor series and common integers (factorial, bernouilli...) appearing in taylor coefficients
01738 namespace aerobus {
01739     namespace internal {
01740         template<typename T, size_t x, typename E = void>
01741         struct factorial {};
01742 
01743         template<typename T, size_t x>
01744         struct factorial<T, x, std::enable_if_t<(x > 0)>> {
01745         private:
01746             template<typename, size_t, typename>
01747             friend struct factorial;
01748         public:
01749             using type = typename T::template mul_t<typename T::template val<x>, typename factorial<T,
```

```
    x - 1>::type>;
01750             static constexpr typename T::inner_type value = type::template get<typename
    T::inner_type>();
01751         };
01752
01753         template<typename T>
01754         struct factorial<T, 0> {
01755          public:
01756             using type = typename T::one;
01757             static constexpr typename T::inner_type value = type::template get<typename
    T::inner_type>();
01758         };
01759     }  // namespace internal
01760
01764     template<typename T, size_t i>
01765     using factorial_t = typename internal::factorial<T, i>::type;
01766
01767     template<typename T, size_t i>
01768     inline constexpr typename T::inner_type factorial_v = internal::factorial<T, i>::value;
01769
01770     namespace internal {
01771         template<typename T, size_t k, size_t n, typename E = void>
01772         struct combination_helper {};
01773
01774         template<typename T, size_t k, size_t n>
01775         struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k <= (n / 2) && k > 0)>> {
01776             using type = typename FractionField<T>::template mul_t<
01777                 typename combination_helper<T, k - 1, n - 1>::type,
01778                 makefraction_t<T, typename T::template val<n>, typename T::template val<k>>;
01779         };
01780
01781         template<typename T, size_t k, size_t n>
01782         struct combination_helper<T, k, n, std::enable_if_t<(n >= 0 && k > (n / 2) && k > 0)>> {
01783             using type = typename combination_helper<T, n - k, n>::type;
01784         };
01785
01786         template<typename T, size_t n>
01787         struct combination_helper<T, 0, n> {
01788             using type = typename FractionField<T>::one;
01789         };
01790
01791         template<typename T, size_t k, size_t n>
01792         struct combination {
01793             using type = typename internal::combination_helper<T, k, n>::type::x;
01794             static constexpr typename T::inner_type value =
01795                         internal::combination_helper<T, k, n>::type::template get<typename
    T::inner_type>();
01796         };
01797     }  // namespace internal
01798
01801     template<typename T, size_t k, size_t n>
01802     using combination_t = typename internal::combination<T, k, n>::type;
01803
01804     template<typename T, size_t k, size_t n>
01805     inline constexpr typename T::inner_type combination_v = internal::combination<T, k, n>::value;
01806
01807     namespace internal {
01808         template<typename T, size_t m>
01809         struct bernouilli;
01810
01811         template<typename T, typename accum, size_t k, size_t m>
01812         struct bernouilli_helper {
01813             using type = typename bernouilli_helper<
01814                 T,
01815                 addfractions_t<T,
01816                     accum,
01817                     mulfractions_t<T,
01818                         makefraction_t<T,
01819                             combination_t<T, k, m + 1>,
01820                             typename T::one>,
01821                         typename bernouilli<T, k>::type
01822                     >
01823                 >,
01824                 k + 1,
01825                 m>::type;
01826         };
01827
01828         template<typename T, typename accum, size_t m>
01829         struct bernouilli_helper<T, accum, m, m> {
01830             using type = accum;
01831         };
01832
01833
01834
01835         template<typename T, size_t m>
01836         struct bernouilli {
01837             using type = typename FractionField<T>::template mul_t<
```

```
01838                 typename internal::bernouilli_helper<T, typename FractionField<T>::zero, 0, m>::type,
01839                 makefraction_t<T,
01840                 typename T::template val<static_cast<typename T::inner_type>(-1)>,
01841                 typename T::template val<static_cast<typename T::inner_type>(m + 1)>
01842                 >
01843             >;
01844 
01845             template<typename floatType>
01846             static constexpr floatType value = type::template get<floatType>();
01847         };
01848 
01849         template<typename T>
01850         struct bernouilli<T, 0> {
01851             using type = typename FractionField<T>::one;
01852 
01853             template<typename floatType>
01854             static constexpr floatType value = type::template get<floatType>();
01855         };
01856     }  // namespace internal
01857 
01861     template<typename T, size_t n>
01862     using bernouilli_t = typename internal::bernouilli<T, n>::type;
01863 
01864     template<typename FloatType, typename T, size_t n >
01865     inline constexpr FloatType bernouilli_v = internal::bernouilli<T, n>::template value<FloatType>;
01866 
01867     namespace internal {
01868         template<typename T, int k, typename E = void>
01869         struct alternate {};
01870 
01871         template<typename T, int k>
01872         struct alternate<T, k, std::enable_if_t<k % 2 == 0» {
01873             using type = typename T::one;
01874             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01875         };
01876 
01877         template<typename T, int k>
01878         struct alternate<T, k, std::enable_if_t<k % 2 != 0» {
01879             using type = typename T::template sub_t<typename T::zero, typename T::one>;
01880             static constexpr typename T::inner_type value = type::template get<typename
      T::inner_type>();
01881         };
01882     }  // namespace internal
01883 
01886     template<typename T, int k>
01887     using alternate_t = typename internal::alternate<T, k>::type;
01888 
01889     template<typename T, size_t k>
01890     inline constexpr typename T::inner_type alternate_v = internal::alternate<T, k>::value;
01891 
01892     // pow
01893     namespace internal {
01894         template<typename T, auto p, auto n>
01895         struct pow {
01896             using type = typename T::template mul_t<typename T::template val<p>, typename pow<T, p, n
      - 1>::type>;
01897         };
01898 
01899         template<typename T, auto p>
01900         struct pow<T, p, 0> { using type = typename T::one; };
01901     }
01902 
01903     template<typename T, auto p, auto n>
01904     using pow_t = typename internal::pow<T, p, n>::type;
01905 
01906     namespace internal {
01907         template<typename, template<typename, size_t> typename, class>
01908         struct make_taylor_impl;
01909 
01910         template<typename T, template<typename, size_t> typename coeff_at, size_t... Is>
01911         struct make_taylor_impl<T, coeff_at, std::integer_sequence<size_t, Is...» {
01912             using type = typename polynomial<FractionField<T»::template val<typename coeff_at<T,
      Is>::type...>;
01913         };
01914     }
01915 
01916     // generic taylor serie, depending on coefficients
01917     template<typename T, template<typename, size_t index> typename coeff_at, size_t deg>
01918     using taylor = typename internal::make_taylor_impl<
01919         T,
01920         coeff_at,
01921         internal::make_index_sequence_reverse<deg + 1»::type;
01922 
01923     namespace internal {
01924         template<typename T, size_t i>
01925         struct exp_coeff {
```

```
01926             using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
01927         };
01928
01929         template<typename T, size_t i, typename E = void>
01930         struct sin_coeff_helper {};
01931
01932         template<typename T, size_t i>
01933         struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
01934             using type = typename FractionField<T>::zero;
01935         };
01936
01937         template<typename T, size_t i>
01938         struct sin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
01939             using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i»;
01940         };
01941
01942         template<typename T, size_t i>
01943         struct sin_coeff {
01944             using type = typename sin_coeff_helper<T, i>::type;
01945         };
01946
01947         template<typename T, size_t i, typename E = void>
01948         struct sh_coeff_helper {};
01949
01950         template<typename T, size_t i>
01951         struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
01952             using type = typename FractionField<T>::zero;
01953         };
01954
01955         template<typename T, size_t i>
01956         struct sh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
01957             using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
01958         };
01959
01960         template<typename T, size_t i>
01961         struct sh_coeff {
01962             using type = typename sh_coeff_helper<T, i>::type;
01963         };
01964
01965         template<typename T, size_t i, typename E = void>
01966         struct cos_coeff_helper {};
01967
01968         template<typename T, size_t i>
01969         struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
01970             using type = typename FractionField<T>::zero;
01971         };
01972
01973         template<typename T, size_t i>
01974         struct cos_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
01975             using type = makefraction_t<T, alternate_t<T, i / 2>, factorial_t<T, i»;
01976         };
01977
01978         template<typename T, size_t i>
01979         struct cos_coeff {
01980             using type = typename cos_coeff_helper<T, i>::type;
01981         };
01982
01983         template<typename T, size_t i, typename E = void>
01984         struct cosh_coeff_helper {};
01985
01986         template<typename T, size_t i>
01987         struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
01988             using type = typename FractionField<T>::zero;
01989         };
01990
01991         template<typename T, size_t i>
01992         struct cosh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
01993             using type = makefraction_t<T, typename T::one, factorial_t<T, i»;
01994         };
01995
01996         template<typename T, size_t i>
01997         struct cosh_coeff {
01998             using type = typename cosh_coeff_helper<T, i>::type;
01999         };
02000
02001         template<typename T, size_t i>
02002         struct geom_coeff { using type = typename FractionField<T>::one; };
02003
02004
02005         template<typename T, size_t i, typename E = void>
02006         struct atan_coeff_helper;
02007
02008         template<typename T, size_t i>
02009         struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
02010             using type = makefraction_t<T, alternate_t<T, i / 2>, typename T::template val<i»;
02011         };
02012
```

```
02013         template<typename T, size_t i>
02014         struct atan_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
02015             using type = typename FractionField<T>::zero;
02016         };
02017
02018         template<typename T, size_t i>
02019         struct atan_coeff { using type = typename atan_coeff_helper<T, i>::type; };
02020
02021         template<typename T, size_t i, typename E = void>
02022         struct asin_coeff_helper;
02023
02024         template<typename T, size_t i>
02025         struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
02026             using type = makefraction_t<T,
02027                 factorial_t<T, i - 1>,
02028                 typename T::template mul_t<
02029                     typename T::template val<i>,
02030                     T::template mul_t<
02031                         pow_t<T, 4, i / 2>,
02032                         pow<T, factorial<T, i / 2>::value, 2
02033                     >
02034                 >
02035                 »;
02036         };
02037
02038         template<typename T, size_t i>
02039         struct asin_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
02040             using type = typename FractionField<T>::zero;
02041         };
02042
02043         template<typename T, size_t i>
02044         struct asin_coeff {
02045             using type = typename asin_coeff_helper<T, i>::type;
02046         };
02047
02048         template<typename T, size_t i>
02049         struct lnp1_coeff {
02050             using type = makefraction_t<T,
02051                 alternate_t<T, i + 1>,
02052                 typename T::template val<i»;
02053         };
02054
02055         template<typename T>
02056         struct lnp1_coeff<T, 0> { using type = typename FractionField<T>::zero; };
02057
02058         template<typename T, size_t i, typename E = void>
02059         struct asinh_coeff_helper;
02060
02061         template<typename T, size_t i>
02062         struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
02063             using type = makefraction_t<T,
02064                 typename T::template mul_t<
02065                     alternate_t<T, i / 2>,
02066                     factorial_t<T, i - 1>
02067                 >,
02068                 typename T::template mul_t<
02069                     T::template mul_t<
02070                         typename T::template val<i>,
02071                         pow_t<T, (factorial<T, i / 2>::value), 2>
02072                     >,
02073                     pow_t<T, 4, i / 2>
02074                 >
02075             >;
02076         };
02077
02078         template<typename T, size_t i>
02079         struct asinh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
02080             using type = typename FractionField<T>::zero;
02081         };
02082
02083         template<typename T, size_t i>
02084         struct asinh_coeff {
02085             using type = typename asinh_coeff_helper<T, i>::type;
02086         };
02087
02088         template<typename T, size_t i, typename E = void>
02089         struct atanh_coeff_helper;
02090
02091         template<typename T, size_t i>
02092         struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 1» {
02093             // 1/i
02094             using type = typename FractionField<T>:: template val<
02095                 typename T::one,
02096                 typename T::template val<static_cast<typename T::inner_type>(i)»;
02097         };
02098
02099         template<typename T, size_t i>
```

```
02100         struct atanh_coeff_helper<T, i, std::enable_if_t<(i & 1) == 0» {
02101             using type = typename FractionField<T>::zero;
02102         };
02103 
02104         template<typename T, size_t i>
02105         struct atanh_coeff {
02106             using type = typename asinh_coeff_helper<T, i>::type;
02107         };
02108 
02109         template<typename T, size_t i, typename E = void>
02110         struct tan_coeff_helper;
02111 
02112         template<typename T, size_t i>
02113         struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
02114             using type = typename FractionField<T>::zero;
02115         };
02116 
02117         template<typename T, size_t i>
02118         struct tan_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
02119         private:
02120             // 4^((i+1)/2)
02121             using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
02122             // 4^((i+1)/2) - 1
02123             using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename
      FractionField<T>::one>;
02124             // (-1)^((i-1)/2)
02125             using altp = typename FractionField<T>::template inject_t<alternate_t<T, (i - 1) / 2»;
02126             using dividend = typename FractionField<T>::template mul_t<
02127                 altp,
02128                 FractionField<T>::template mul_t<
02129                 _4p,
02130                 FractionField<T>::template mul_t<
02131                 _4pm1,
02132                 bernouilli_t<T, (i + 1)>
02133                 >
02134                 >
02135             >;
02136         public:
02137             using type = typename FractionField<T>::template div_t<dividend,
02138                 typename FractionField<T>::template inject_t<factorial_t<T, i + 1»>;
02139         };
02140 
02141         template<typename T, size_t i>
02142         struct tan_coeff {
02143             using type = typename tan_coeff_helper<T, i>::type;
02144         };
02145 
02146         template<typename T, size_t i, typename E = void>
02147         struct tanh_coeff_helper;
02148 
02149         template<typename T, size_t i>
02150         struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) == 0» {
02151             using type = typename FractionField<T>::zero;
02152         };
02153 
02154         template<typename T, size_t i>
02155         struct tanh_coeff_helper<T, i, std::enable_if_t<(i % 2) != 0» {
02156         private:
02157             using _4p = typename FractionField<T>::template inject_t<pow_t<T, 4, (i + 1) / 2»;
02158             using _4pm1 = typename FractionField<T>::template sub_t<_4p, typename
      FractionField<T>::one>;
02159             using dividend =
02160                 typename FractionField<T>::template mul_t<
02161                 _4p,
02162                 typename FractionField<T>::template mul_t<
02163                 _4pm1,
02164                 bernouilli_t<T, (i + 1)>
02165                 >
02166                 >::type;
02167         public:
02168             using type = typename FractionField<T>::template div_t<dividend,
02169                 FractionField<T>::template inject_t<factorial_t<T, i + 1»>;
02170         };
02171 
02172         template<typename T, size_t i>
02173         struct tanh_coeff {
02174             using type = typename tanh_coeff_helper<T, i>::type;
02175         };
02176     }  // namespace internal
02177 
02181     template<typename T, size_t deg>
02182     using exp = taylor<T, internal::exp_coeff, deg>;
02183 
02187     template<typename T, size_t deg>
02188     using expm1 = typename polynomial<FractionField<T»::template sub_t<
02189         exp<T, deg>,
02190         typename polynomial<FractionField<T»::one>;
```

```
02191
02195     template<typename T, size_t deg>
02196     using lnp1 = taylor<T, internal::lnp1_coeff, deg>;
02197
02201     template<typename T, size_t deg>
02202     using atan = taylor<T, internal::atan_coeff, deg>;
02203
02207     template<typename T, size_t deg>
02208     using sin = taylor<T, internal::sin_coeff, deg>;
02209
02213     template<typename T, size_t deg>
02214     using sinh = taylor<T, internal::sh_coeff, deg>;
02215
02219     template<typename T, size_t deg>
02220     using cosh = taylor<T, internal::cosh_coeff, deg>;
02221
02225     template<typename T, size_t deg>
02226     using cos = taylor<T, internal::cos_coeff, deg>;
02227
02231     template<typename T, size_t deg>
02232     using geometric_sum = taylor<T, internal::geom_coeff, deg>;
02233
02237     template<typename T, size_t deg>
02238     using asin = taylor<T, internal::asin_coeff, deg>;
02239
02243     template<typename T, size_t deg>
02244     using asinh = taylor<T, internal::asinh_coeff, deg>;
02245
02249     template<typename T, size_t deg>
02250     using atanh = taylor<T, internal::atanh_coeff, deg>;
02251
02255     template<typename T, size_t deg>
02256     using tan = taylor<T, internal::tan_coeff, deg>;
02257
02261     template<typename T, size_t deg>
02262     using tanh = taylor<T, internal::tanh_coeff, deg>;
02263 }  // namespace aerobus
02264
02265 // continued fractions
02266 namespace aerobus {
02269     template<int64_t... values>
02270     struct ContinuedFraction {};
02271
02272     template<int64_t a0>
02273     struct ContinuedFraction<a0> {
02274         using type = typename q64::template inject_constant_t<a0>;
02275         static constexpr double val = type::template get<double>();
02276     };
02277
02278     template<int64_t a0, int64_t... rest>
02279     struct ContinuedFraction<a0, rest...> {
02280         using type = q64::template add_t<
02281                 typename q64::template inject_constant_t<a0>,
02282                 typename q64::template div_t<
02283                     typename q64::one,
02284                     typename ContinuedFraction<rest...>::type
02285                 »;
02286         static constexpr double val = type::template get<double>();
02287     };
02288
02293     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>;
02296     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>;
02298     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>;
02300     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, 1, 2>;
      // NOLINT
02301 }  // namespace aerobus
02302
02303 // known polynomials
02304 namespace aerobus {
02305     namespace internal {
02306         template<int kind, int deg>
02307         struct chebyshev_helper {
02308             using type = typename pi64::template sub_t<
02309                 typename pi64::template mul_t<
02310                     typename pi64::template mul_t<
02311                         pi64::inject_constant_t<2>,
02312                         typename pi64::X
02313                     >,
02314                     typename chebyshev_helper<kind, deg-1>::type
02315                 >,
02316                 typename chebyshev_helper<kind, deg-2>::type
02317             >;
02318         };
```

```
02319
02320         template<>
02321         struct chebyshev_helper<1, 0> {
02322             using type = typename pi64::one;
02323         };
02324
02325         template<>
02326         struct chebyshev_helper<1, 1> {
02327             using type = typename pi64::X;
02328         };
02329
02330         template<>
02331         struct chebyshev_helper<2, 0> {
02332             using type = typename pi64::one;
02333         };
02334
02335         template<>
02336         struct chebyshev_helper<2, 1> {
02337             using type = typename pi64::template mul_t<
02338                             typename pi64::inject_constant_t<2>,
02339                             typename pi64::X>;
02340         };
02341     }  // namespace internal
02342
02345     template<size_t deg>
02346     using chebyshev_T = typename internal::chebyshev_helper<1, deg>::type;
02347
02350     template<size_t deg>
02351     using chebyshev_U = typename internal::chebyshev_helper<2, deg>::type;
02352 }  // namespace aerobus
02353
02354 #endif // __INC_AEROBUS__ // NOLINT
```

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