ACS

Algorithms for Complex Shapes
with Certified Numerics and Topology

Traits classes for polynomial gcd computation over algebraic extensions

Michael Hemmer Dominik Hülse

ACS Technical Report No.: ACS-TR-241405-03

Part of deliverable: WP-III/D4
Site: MPI
Month: 24

Project co-funded by the European Commission within FP6 (2002–2006)
under contract nr. IST-006413

Traits classes for polynomial gcd computation over algebraic extensions

Michael Hemmer Dominik Hülse

Abstract

Reported are experiences on generic implementations for polynomial gcd computation over algebraic extensions within EXACUS. In particular we report on preliminary benchmarks for gcd computation of univariate polynomials comparing the naive approach with preliminary implementations based on modular arithmetic. The report includes specification of traits classes needed to deal with algebraic coefficients. The presented traits classes are supposed to serve as a design study in order to extend the current framework in the algebraic foundations package of CGAL.

1 Introduction

CGAL is targeting towards exact computation with non-linear objects, in particular objects defined on algebraic curves and surfaces. As a consequence arithmetic on algebraic extensions plays a more important role in related implementations. Within the ACS project this is the case for the approaches targeting towards the arrangement of quadrics [6, 1]. For the approach based on the exact parameterization of the appearing intersection curves of quadrics, it turned out that the *gcd* computation for univariate polynomials over algebraic extensions of degree 2 is currently the main bottleneck. For these problems, methods based on modular arithmetic are known to be very efficient. But to the best of our knowledge, there is no generic open source code available, that supports a modular gcd for polynomials over algebraic extensions. From there, this report is focused on a generic implementation of modular methods for univariate polynomials over algebraic extensions of degree 2. However, the presented traits classes, which have been designed to support the investigated modular algorithms, are meant to support general algebraic extensions. Moreover, it should be straight forward to extend the implementation to true multivariate modular gcd, once a generic univariate implementation is established.

The report is structured as follows: Section 2 provides an overview of the investigated algorithms based on modular methods. Section 3 provides details on the implementation and presents the introduced traits classes. In section 4 on benchmarks, we compare the new implementation with the naive approach as it was implemented in EXACUS so far. Section 5 will draw conclusions and will outline further work. The documentation of the implemented traits classes is included in the appendix.

2 The Modular Methods

We now recall the most fundamental modular methods of interest. Brown's modular GCD algorithm [2] solved the problem for polynomials in $\mathbb{Z}[x_1,\ldots,x_n]$. Langemyr and McCallum developed in [7] an algorithm to compute the gcd $g(x)$ of two univariate polynomials $f_1(x),\ f_2(x)$ over an algebraic extension. A further improvement was Encarnacion's modular GCD algorithm in [3]. The main advantage over the modular algorithm of Langemyr and McCallum is, that it uses the rational reconstruction developed by Wang in [12]. From there, Encarnacion's algorithm is output sensitive, that is, the number of primes needed depends on the size of the rational coefficients in the output $g(x)$ and not on bounds based on the size of the inputs $f_1(x),\ f_2(x)$. A second advantage of using rational reconstruction is that it does not require a denominator bound, that is, a multiple of the LCM of the denominators of the rational coefficients appearing in $g(x)$.

2.1 The Modular GCD Algorithm over the Integers

In this section we outline the modular GCD algorithm for univariate polynomials with integer coefficients. For a more details study [2] section 4.3.

Let $\mathbb{Z}$ denote the domain of integers, and $\mathbb{Z}_p$ denote the field of integers modulo a prime p. Let F_1' and F_2' be given nonzero polynomials in $\mathbb{Z}[x]$. Algorithm M computes their GCD G', making essential use of the GCD Algorithm P[1] for polynomials over fields.

Algorithm M:[2]

(1) Set $c_1 = cont(F_1'),\ c_2 = cont(F_2'), c = gcd(c_1, c_2)$
(2) Set $F_1 = F_1'/c_1,\ F_2 = F_2'/c_2$
(3) Set $f_1 = lc(F_1),\ f_2 = lc(F_2),\ \overline{g} = gcd(f_1, f_2)$.
(4) Set n = 0, e = min(deg(F_1),deg(F_2))
(5) Let p be a new odd prime not dividing f_1 or f_2
(6) Set $\tilde{g} = \overline{g}\ mod\ p, \tilde{F_1} = F_1\ mod\ p, \tilde{F_2} = F_2\ mod\ p$
(7) Invoke Algorithm P to compute $\tilde{G} = \tilde{g}\cdot\ gcd(\tilde{F_1},\tilde{F_2})$, over $\mathbb{Z}_p[x]$
(8) If deg($\tilde{G}$) = 0, set G = 1, and skip to step (15).
If deg($\tilde{G}$) $>$ e, p is an unlucky prime, so go back to step (5).
If deg($\tilde{G}$) $<$ e, a former prime was unlucky, set n = 0, e = deg($\tilde{G}$)
(9) Set n = n+1
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*).
$(q, G^*) := chinese\_remainder((q, G^*),(p, \tilde{G}))$
(12) If the coefficients of G^* have changed go back to step (5).
(13) If $G^* \nmid \overline{g}\cdot F_1$ or $G^* \nmid \overline{g}\cdot F_2$ go back to step (5)
(14) Set G = pp(G^*)
(15) Set G' = cG, and return.

---

[1] For the univariate case algorithm P is available within EXACUS

[2] We use the following notation: $cont(F)$: The (unit normal) *gcd* of the coefficients of F. $pp(F)$: The primitive part of F, i.e. $F/cont(F)$. $lc(F)$: The leading coefficient of F. $monic(F)$: The monic associate to F, i.e. $F/lc(F)$.

2.2 Modular GCD Algorithms over Algebraic Extensions

In this section we want to discuss two modular gcd algorithms for univariate polynomials over algebraic extensions. Since most algorithms in EXACUS are designed as fraction free algorithms we aim for a fraction free algorithm.

Let D denote some algebraic extension of $\mathbb{Z}$, and L the quotient field of D. Let F_1' and F_2' be given nonzero polynomials in $D[x]$. We are only interested in computing the *gcd* of F_1' and F_2' up to a constant factor, since in general D is not a Euclidean Ring.

2.2.1 Fraction Free Algorithm

The following algorithm **F** computes the *gcd* of two non zero polynomials F_1' and $F_2' \in D[x]$ up to a constant factor, without introducing any fractions. It is motivated by the method of Langemyr and McCallum presented in [7]. In principal, Algorithm **F** is just a variant of algorithm M. The most important differences are steps (1) to (3a). First, the polynomials F_1 and F_2 are multiplied by a normalization factor, such that the leading coefficients are in $\mathbb{Z}$. Step (2) removes superfluous scalar factors. The most important step is (3a), in which $\overline{g}$ is extended by the denominator bound of D, as it is given in [7, 3]. From there, it is guaranteed that $\tilde{G}$ is associated to a fraction free representative of $gcd(F_1,F_2)$ and that the Chinese remainder finally succeeds in computing the $gcd \in D[x]$.

Algorithm F:[3]

(1) Set $F_1 = normalization\_factor(lc(F_1')) * F_1'$
Set $F_2 = normalization\_factor(lc(F_2')) * F_2'$
(2) Set $F_1 = F_1/scalar\_factor(F_1)$
Set $F_2 = F_2/scalar\_factor(F_2)$
(3) Set $f_1 = lc(F_1),\ f_2 = lc(F_2)$ // $f_1, f_2 \in \mathbb{Z}$ due to step (1)
Set $\overline{g} = gcd(f_1, f_2)$.
(3a) Set $\overline{g} = \overline{g} * denominator\_for\_algebraic\_integers(D)$.
(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$
(5) Let p be a new odd prime not dividing f_1 or f_2
(6) Set $\tilde{g} = \overline{g}\ mod\ p$, $\tilde{F_1} = F_1\ mod\ p$, $\tilde{F_2} = F_2\ mod\ p$
(7) Invoke Algorithm P to compute $\tilde{G} = \tilde{g} \cdot\ gcd(\tilde{F_1}, \tilde{F_2})$, over $D_p[x]$
(8) If $deg(\tilde{G}) = 0$, set $G^* = G = 1$, and skip to step (14).
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, a former prime was unlucky, set $n = 0$, $e = deg(\tilde{G})$
(9) Set $n = n+1$
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*).
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$
(12) If the coefficients of G^* have changed go back to step (5).
(13) If $G^* \nmid F_1$ or $G^* \nmid F_2$ go back to step (5) // '$\nmid$' denotes pseudo division
(14) return G^*

[3] For $scalar\_factor(F)$, $normalization\_factor(x)$ and $denominator\_for\_algebraic\_integer(D)$ we refer to section 3.

In order to illustrate the role of the denominator bound within algorithm **F**, we illustrate its behavior without step (3a) in the following example:
Let $G = ((1+\sqrt{5})x+2)$ be the common factor of

$$H_1 = ((1+\sqrt{5})x+2)\cdot((1-\sqrt{5})x+4) = 4x^2 - 2(3+\sqrt{5})x - 8 \in D[x],$$
$$H_2 = ((1+\sqrt{5})x+2)\cdot((1-\sqrt{5})x+6) = 4x^2 - 4(2+\sqrt{5})x - 12 \in D[x].$$

However, let the input of algorithm **F** be:

$$F_1' = H_1/2 = 2x^2 - (3+\sqrt{5})x - 4 \in D[x]$$

$$F_2' = H_2/4 = x^2 + (3+\sqrt{5})x - 3 \in D[x]$$

Now note, that without step (3a), $\overline{g}$ is set to $gcd(lc(F_1), lc(F_2)) = 1$. From there, $\tilde{G}$ is associated to

$$monic(G) = x + 1/2 - \sqrt{5}/4 \in L[x]$$

and the algorithm will not terminate since the Chinese remainder is able to reconstruct polynomials in $D[x]$ only.

The multiplicative denominator bound depends on the concrete algebraic extension that D represents. For more details we refer to [11, 10, 8]. However, in case of algebraic extensions of degree 2 this is just 4 times the discriminant of the minimal polynomial, which in this example is: $4\cdot 5 = 20$.

2.2.2 Algorithm using Wang's Rational Reconstruction

As algorithm **F**, the following algorithm **W** computes the *gcd* of two non zero polynomials F_1' and $F_2' \in D[x]$ up to a constant factor.
Since algorithm **F** uses Wang's rational reconstruction algorithm, lets recall the functionality of this algorithm. For given integers $m,\ u$ it tries to reconstruct $n,\ d \in \mathbb{Z}$ s.t. $gcd(n,d) = 1$ and $n/d \equiv u \; mod \; m$. However, we can rely on, if m is sufficiently large, the algorithm works with high probability. For more information see [12].
Algorithm **W** modifies **F** by skipping step (3a) of algorithm **F**. Thus, it is not guaranteed that the Chinese remainder will finally succeed in recovering the *gcd*, since $\tilde{G}$ may be associated to a polynomial containing fractions. However, due to the fact that $\tilde{G}$ is yet obtained by $\tilde{g}\cdot\ gcd(\tilde{F_1},\tilde{F_2})$ in step (7), it is still likely that $\tilde{G}$ is associated to a polynomial in $D[x]$. From there, the use of Wang's algorithm in step (13) can be seen as a fall back, for the unlikely case that the multiplicative denominator bound was really needed.

Algorithm **W**:

(1) Set $F_1 = normalization\_factor(lc(F_1')) * F_1'$
Set $F_2 = normalization\_factor(lc(F_2')) * F_2'$
(2) Set $F_1 = F_1/scalar\_factor(F_1)$
Set $F_2 = F_2/scalar\_factor(F_2)$
(3) Set $f_1 = lc(F_1),\ f_2 = lc(F_2)$ // $f_1, f_2 \in \mathbb{Z}$ due to step (1)
Set $\overline{g} = gcd(f_1, f_2)$.
// we **skip step (3a)** of algorithm **F**

(4) Set $n = 0$, $e = min(deg(F_1), deg(F_2))$
(5) Let p be a new odd prime not dividing f_1, f_2 and $denominator\_for\_algebraic\_integers(D)$
(6) Set $\tilde{g} = \overline{g}\ mod\ p$, $\tilde{F_1} = F_1\ mod\ p$, $\tilde{F_2} = F_2\ mod\ p$
(7) Invoke Algorithm P to compute $\tilde{G} = \tilde{g} \cdot\ gcd(\tilde{F_1}, \tilde{F_2})$, all in $D_p[x]$
(8) If $deg(\tilde{G}) = 0$, set $G = 1$, and skip to step (14).
If $deg(\tilde{G}) > e$, p is an unlucky prime, so go back to step (5).
If $deg(\tilde{G}) < e$, a former prime was unlucky, set $n = 0$, $e = deg(\tilde{G})$
(9) Set $n = n + 1$
(10) If $n = 1$, set the tuple $(q, G^*) = (p, \tilde{G})$ and go back to step (5).
(11) Use Chinese remainder to update the tuple (q, G^*).
$(q, G^*) := chinese\_remainder((q, G^*), (p, \tilde{G}))$
(12) If the coefficients of G^* have not changed and
if $G^* \mid F_1$ and $G^* \mid F_2$ return G^* // '$\mid$' denotes pseudo division
(13) Try Wang's rational reconstruction to obtain $G'^* := wang(G^*)$.
If Wang's algorithm fails go back to step(5).
(14) If $G'^* \nmid F_1$ or $G'^* \nmid F_2$ go back to step (5). // '$\nmid$' denotes pseudo division
(15) return G'^*

3 Implementation

All algorithms have been implemented within EXACUS. However, due to the integration of the most fundamental parts of EXACUS into CGAL, i.e. the Algebraic Foundations package [5] and the package on Modular Arithmetic [4], an integration into CGAL should be straight forward.

We introduced several traits classes to provide the functionality needed by the algorithms presented in section 2. A detailed documentation of the traits classes is provided in the appendix of this report.

Scalar Factor Traits

The traits class, `Scalar_factor_traits`, is of general interest, since a recurring problem in handling compound types, as polynomial, homogeneous vectors or matrices, is how to remove superfluous factors even though the coefficient type does not provide a *gcd*. From there, the scalar factor is in general not the gcd over the coefficients, instead the *gcd* of the scalar factors of all coefficients is returned.
This is in particular useful for types representing algebraic extensions of $\mathbb{Z}$, e.g. the scalar factor of a one-root-number $a + b\sqrt{c}$, with $a, b, c \in \mathbb{Z}$, is $gcd(a, b)$.

Chinese Remainder Traits

The traits class, `Chinese_remainder_traits`, provides access to the Chinese remainder algorithm. The traits class is designed such that it is applicable to compound types that can be decomposed into scalar coefficients in the sense of the `Scalar_factor_traits`, i.e. the actual algorithm is applied to each scalar coefficient. For example, a Chinese remainder for a polynomial calls Chinese remainder for each coefficient and returns the resulting polynomial.

Wang Traits

The traits class, `Wang_traits`, provides access to Wang's algorithm. The design of the traits class is symmetric to the `Chinese_remainder_traits` and is applicable to all types that are decomposable into scalar coefficients in the sense of the `Scalar_factor_traits`.

Algebraic Extension Traits

The traits class, `Algebraic_extension_traits`, provides the the functionality related to algebraic extensions:

- $normalization\_factor(x)$
  For a given $x \in D$ compute a factor a such that $x \cdot a \in \mathbb{Z}$.
  This is used in algorithm **F** and **W** to modify the input polynomials such that their leading coefficients are in $\mathbb{Z}$.

- $denominator\_for\_algebraic\_integers$
  This function computes the multiplicative denominator bound as discussed in 2.

4 Benchmarks

To analyse the impact of coefficient bit-size, we generated two families of instances of univariate random polynomials with growing bit-size for integer coefficients (INT_B) and algebraic extensions of degree 2 (EXT_B). Each instance contains 50 pairs of random polynomials of degree 8. Each pair is composed of three factors, a GCD of degree 1 and the two co-factors of degree 7. The polynomials are randomized in the sense, that each coefficient has been diced, such that their integer parts have the desired bit-size.

Since the degree of the contained GCD has an impact on the length of the polynomial remainder sequence and thus on the runtime of the algorithms, we generated two more families of instances of growing degree of the GCD. One family for integer coefficients (INT_D) and a second with algebraic extensions of degree 2 (EXT_D). Again, each instance contains 50 randomized polynomial pairs with exact degree 8. Each pair is composed of three factors. The GCD with the desired degree and the two co-factors. Each factor is randomized in the sense, that each coefficient has been diced, such that their integer parts have exactly 1000 bits.

The benchmarks were measured on a Pentium(R) M processor 1.7 GHz with 512 KB cache under Linux and the GNU C++ compiler v3.4.6 with optimizations (`-O3`) and disabled assertions (`-DNDEBUG`).

Figure 1 shows, that the algorithm **M** performs far better then the old, non-modular implementation. The same picture can be found in figure 2 for the algorithms **F** and **W**. Moreover, it shows that the original version of algorithm **W**, as presented in section 2, is slower than algorithm **F**. This is because the original version of **W** invokes Wang's algorithm within each loop. But since Wang's algorithm is only a fall back, we modified algorithm **W** such that Wang's algorithm is invoked for every twentieth prime only. Figure 2 shows, that this modified version performs better than algorithm

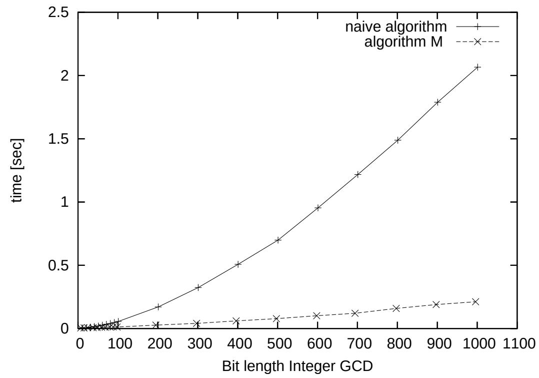

Figure 1: Algorithm **M** compared to the old gcd approach on family (INT_B).

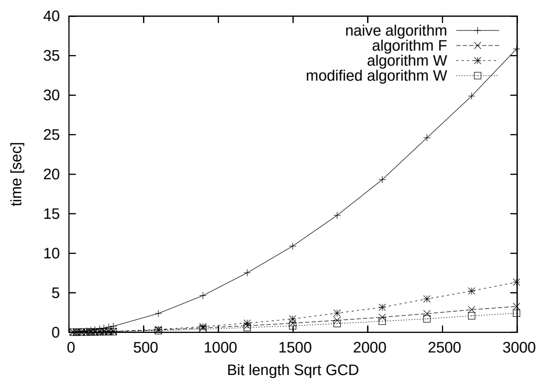

Figure 2: Old gcd approach compared to algorithm **F** and **W** on family (EXT_B).

F. This is due to the fact that the extension of $\overline{g}$, step (3a), by the multiplicative denominator bound is in fact not needed for the random instances. From there algorithm **F** is slower than the modified algorithm **W**, since it needs more primes due to the overhead in the coefficient length.

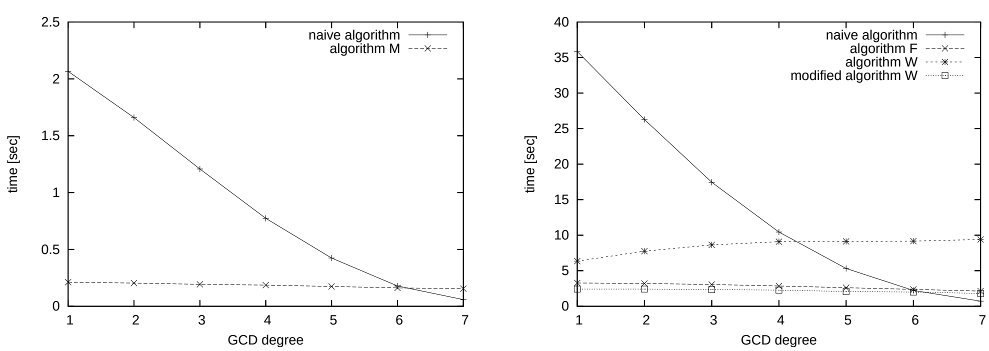

Figure 3: To the left, algorithm **M** compared to the old implementation on family (INT_D). To the right, algorithm **F** and **W** compared to the old implementation on family (EXT_D).

Figure 3 shows, that the performance of the modular methods is almost independent from the degree of the final GCD. This is caused by the fact, that most of the time is spent within the extended Euclidean algorithm of the Chinese remainder. But since, within each round, we cash the result of the extended Euclidean algorithm, this is almost independent from the number of coefficients. On the other hand we see, that due to the decreasing length of the polynomial remainder sequence, the old method finally performs better than the modular methods.

5 Conclusions and Further Work

We see the current implementation for algebraic extensions of degree 2, as a proof of concept for the presented traits classes. We propose to integrate the presented traits classes into the current framework of the ALGEBRAIC FOUNDATIONS package [5] in CGAL. However, there are some redundancies in the design, e.g. the traits classes related to scalar factor, Chinese remainder and Wang's algorithm could be combined into one traits class related to scalar coefficients.

Moreover, it is obvious that we should aim for a true modular multivariate gcd, as it is presented in [2]. This requires a modification of algorithm **P**. But this is supposed to be independent from the coefficient type, since this is a multivariate *gcd* over some field, e.g. $\mathbb{Z}_p$ or D_p.

We also expect some improves for algorithm **W**, since the current implementation of Wang's algorithm does not take advantage of the known multiplicative denominator bound, as it is indicated in [9]. Moreover we should explore other methods for rational reconstruction as presented in [9].

The result in the benchmarks show, that the modular methods perform far better than the old non-modular implementation within EXACUS. From there, we expect that these results will have a great positive impact on other approaches within the ACS projects, i.e. those targeting towards the arrangement of quadrics [6, 1].

We propose to integrate algorithm **M** and **W** into CGAL, where algorithm **M** is used for integer coefficients and algorithm **W** for algebraic extensions.

5.1 Acknowledgment

We thank Arno Eigenwillig, who had made insightful comments on the treatment of ring extensions in EXACUS; they helped to get this work started.

References

[1] Eric Berberich and Michal Meyerovitch. Computing envelopes of quadrics. Technical Report ACS-TR-241402-03, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[2] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18:478–504, 1971.

[3] M. J. Encarnacion. Computing gcds of polynomials over algebraic number fields. *J. Symbolic Computation*, 20:299–313, 1995.

[4] Michael Hemmer. Cgal package for modular arithmetic operations. Technical Report ACS-TR-243406-01, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[5] Michael Hemmer. Cgal package on algebraic foundations. Technical Report ACS-TR-243400-04, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[6] Michael Hemmer, Elmar Schömer, Laurent Dupont, and Sylvain Petitjean. Complete, exact and efficient implementation for computing the adjacency graph of an arrangment of quadrics. Technical Report ACS-TR-241402-01, Algorithms for Complex Shapes with certified topology and numerics, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[7] S. McCallum L. Langemyr. The computation of polynomial gcd's over an algebraic number field. *J. Symbolic Computation*, 8:429–448, 1989.

[8] S. Landau. Factoring polynomials over algebraic number fields. *SIAM Journal on Computing*, 14:184–195, 1985.

[9] M. Monagan. Maximal quotient rational reconstruction: An almost optimal algorithm for rational reconstruction. *ACM Press: New York*, pages 243–249, 2004.

[10] L. P. Rothschild P. J. Weinberger. Factoring polynomials over algebraic number fields. *ACM Transactions on Mathematical Software*, 2.

[11] P. S. Wang. Factoring multivariate polynomials over algebraic number fields. *Math. Comb.*, 30:1215–1231, 1978.

[12] P. S. Wang. A p-adic algorithm for univariate partial fractions. *Proceedings of SYMSAC '81, ACM Press*, pages 212–217, 1981.

Appendix
Reference Manual

Note that the presented traits classes are documented for CGAL.

0.1 Classified Reference Pages

Scalar Factor Traits

Algebraic Extension Traits

Chinese Remainder Traits

Wang Traits

0.2 Alphabetical List of Reference Pages

ScalarFactorTraits

Definition

A model of *ScalarFactorTraits* is associated to an algebraic structure, *Type*.
The traits associates this type to a *Scalar*, which is a model of *EuclideanRing*.

Types

A model of *ScalarFactorTraits* is supposed to provide:

ScalarFactorTraits:: Type	The associated algebraic structure.
ScalarFactorTraits:: Scalar	The associated scalar type.

Functors

ScalarFactorTraits:: Scalar_factor	A model of *ScalarFactorTraits::ScalarFactor*
ScalarFactorTraits:: Scalar_division	A model of *ScalarFactorTraits::ScalarDivision*

Has Models

CGAL::Scalar_factor_traits<T> . . . . page 6

Functor
Concept

ScalarFactorTraits::ScalarFactor

Definition

AdaptableUnaryFunction computes the scalar factor, that is contained in the argument.

Refines

AdaptableUnaryFunction

Types

ScalarFactorTraits::ScalarFactor:: result_type

Is *ScalarFactorTraits::Scalar*.

ScalarFactorTraits::ScalarFactor:: argument_type

Is *ScalarFactorTraits::Type*.

Operations

result_type   *scalar_factor(argument_type x)*

Computes the scalar factor contained within x.

See Also

ScalarFactorTraits . . . . page 3

ScalarFactorTraits::ScalarDivision

Definition

This *AdaptableBinaryFunction* provides integral division by a scalar factor.

Refines

AdaptableUnaryFunction

Types

ScalarFactorTraits::ScalarDivision:: result_type

Is *ScalarFactorTraits::Type*.

ScalarFactorTraits::ScalarDivision:: first_argument_type

Is *ScalarFactorTraits::Type*.

ScalarFactorTraits::ScalarDivision:: second_argument_type

Is *ScalarFactorTraits::Scalar*.

Operations

result_type *scalar_division(first_argument_type x, second_argument_type s)*

returns x/s.
Precondition: s is a scalar factor of x.

See Also

ScalarFactorTraits . . . . page 3

Class

CGAL::Scalar_factor_traits<T>

Definition

An instance of *Scalar_factor_traits<T>* is a model of *ScalarFactorTraits*, where T is the associated type.

#include <CGAL/Scalar_factor_traits.h>

Is Model for the Concepts

ScalarFactorTraits . . . . page 3

AlgebraicExtensionTraits

Concept

Definition

A model of *AlgebraicExtensionTraits* is associated to an algebraic structure, *Type*.
The traits indicates whether the associated type represents an algebraic extension and provides related functionality.

Types

A model of *AlgebraicExtensionTraits* is supposed to provide:

AlgebraicExtensionTraits:: Type — The associated algebraic structure.

AlgebraicExtensionTraits:: Is_algebraic_extension — Boolean tag indicating whether the type is an algebraic extension.
This is either *CGAL::Tag_true* or *CGAL::Tag_false*.

Functors

AlgebraicExtensionTraits:: Normalization_factor — A model of *AlgebraicExtensionTraits::NormalizationFactor*.

AlgebraicExtensionTraits:: Denominator_for_algebraic_integers — A model of *AlgebraicExtensionTraits::DenominatorForAlgebraicIntegers*.

Has Models

CGAL::Algebraic_extension_traits<T> . . . . page 10

Functor
Concept

AlgebraicExtensionTraits::NormalizationFactor

Definition

For a given $x \in R(\alpha)$ this *AdaptableUnaryFunction* computes a factor a, such that for $a \cdot x \in R$.

Refines

AdaptableUnaryFunction

Types

AlgebraicExtensionTraits::NormalizationFactor:: result_type

Is *AlgebraicExtensionTraits::Type*.

AlgebraicExtensionTraits::NormalizationFactor:: argument_type

Is *AlgebraicExtensionTraits::Type*.

Operations

result_type   *normalization_factor(argument_type x)*

Computes normalization factor of x.

See Also

AlgebraicExtensionTraits . . . . page 7

AlgebraicExtensionTraits::DenominatorForAlgebraicIntegers

Definition

AdaptableUnaryFunction, computing a multiplicative bound for the denominator of rational numbers appearing in the representation of the *gcd* of two polynomials, even though the coefficients of the polynomials are algebraic integers.

Since the algebraic extension generated by one coefficient may be only a proper subset of the algebraic extension generated by all coefficients the functor provides a second operator taking an iterator range of coefficients.

Refines

AdaptableUnaryFunction

Types

AlgebraicExtensionTraits::DenominatorForAlgebraicIntegers:: result_type

Is *AlgebraicExtensionTraits::Type*.

AlgebraicExtensionTraits::DenominatorForAlgebraicIntegers:: argument_type

Is *AlgebraicExtensionTraits::Type*.

Operations

result_type *fo(argument_type x)*

Computes the denominator for the algebraic extension generated by *x*.

template < class InputIterator >
result_type *fo(InputIterator begin, InputIterator end)*

Computes the bound for the algebraic extension generated by all coefficients within the iterator range.
Precondition: The value type of InputIterator is *AlgebraicExtensionTraits::Type*

See Also

AlgebraicExtensionTraits . . . . page 7

Class

CGAL::Algebraic_extension_traits<T>

Definition

An instance of *Algebraic_extension_traits<T>* is a model of *AlgebraicExtensionTraits*, where T is the associated type.

#include <CGAL/Algebraic_extension_traits.h>

Is Model for the Concepts

AlgebraicExtensionTraits . . . . page 7

WangTraits

Concept

Definition

A model of *WangTraits* is associated to an algebraic structure, *Type*. It provides rational reconstruction according to Wang's algorithm for the associated type.

Note that the provided functionality is a generalized version of Wang's algorithm, i.e., the associated type is not required to be a *EuclideanRing*. Instead the algorithm is applied to the inner scalar coefficients of the type[1].

Types

A model of *WangTraits* is supposed to provide:

WangTraits:: Type	The associated algebraic structure.
WangTraits:: Scalar	The scalar type, used to represent the denominator. This is a model of *EuclideanRing*.

Functors

WangTraits:: Wang	A model of *WangTraits::Wang*.

Has Models

CGAL::Wang_traits<T> . . . . page 13

[1]Note that this is related to *ScalarFactorTraits*, i.e., the provided scalar types are the same.

WangTraits::Wang

Definition

AdaptableFunctor computing rational reconstruction according to Wang's algorithm.

For given pair $(u,m) \in R$, where R is an *EuclideanRing*, Wang's algorithm tries to compute (n,d) such that $u = n/d \ mod \ m$. Note that the functor provides a generalized in the sense that *WangTraits::Type* is not required to be a model of *EuclideanRing*. Instead the algorithm is applied to the inner scalar coefficients of the type.

Refines

AdaptableFunctor

Types

WangTraits::Wang:: result_type	Is *bool*.

Operations

result_type *wang.operator()(WangTraits::Type u,*
WangTraits::Scalar m,
WangTraits::Type & n,
WangTraits::Scalar& d)

For a given u and m, the function tries to compute numerator n and denominator d via Wang's algorithm. It returns *true* in case the algorithm was successful. Note: n and d are undefined in case the algorithm fails.

See Also

WangTraits . . . . page 11

CGAL::Wang_traits<T>

Definition

An instance of *Wang_traits<T>* is a model of *WangTraits*, where T is the associated type.

#include <CGAL/Wang_traits.h>

Is Model for the Concepts

WangTraits . . . . page 11

ChineseRemainderTraits

Definition

A model of *ChineseRemainderTraits* is associated to an algebraic structure, *Type*. The traits provides access to the Chinese remainder algorithm for the associated type.

Note that the provided Chinese remainder is a generalized version, i.e., the associated type is not required to be a *EuclideanRing*. Instead the Chinese remainder is applied to the inner scalar coefficients[2].

Types

A model of *ChineseRemainderTraits* is supposed to provide:

ChineseRemainderTraits:: Type	The associated algebraic structure.
ChineseRemainderTraits:: Scalar	The scalar type, the actual Chinese remainder is applied to.

Functors

ChineseRemainderTraits:: Chinese_remainder	A model of *ChineseRemainderTraits::ChineseRemainder*.

Has Models

CGAL::Chinese_remainder_traits<T> . . . . page 16

[2] Note that this is related to *ScalarFactorTraits*, i.e., the provided scalar types are the same.

ChineseRemainderTraits::ChineseRemainder

Functor
Concept

Definition

AdaptableFunction computing the Chinese remainder.

Note that the provided Chinese remainder is a generalized version, i.e., the associated type is not required to be a *EuclideanRing*. Instead the Chinese remainder is applied to the inner scalar coefficients[3].

Refines

AdaptableUnaryFunction

Types

typedef void *result_type;*

Operations

result_type *fo(argument_type x)*

For given pairs $(u1, m1)$ and $(u2, m2)$, this function computes the the pair (u, m) such that.
$m = m1 * m2$
$u = u1\ mod\ m1$
$u = u2\ mod\ m2$

Precondition: $gcd(m1, m2) = 1$

See Also

ChineseRemainderTraits . . . . page 14

[3]Note that this is related to *ScalarFactorTraits*, i.e., the provided scalar types are the same.

Class

CGAL::Chinese_remainder_traits<T>

Definition

An instance of *Chinese_remainder_traits<T>* is a model of *ChineseRemainderTraits*, where T is the associated type.

#include <CGAL/Chinese_remainder_traits.h>

Is Model for the Concepts

ChineseRemainderTraits . . . . page 14