YalAA
Yet Another Library for Affine Arithmetic[*]

Stefan Kiel
Department of Computer Science and Applied Cognitive Science - University of Duisburg-Essen
kiel@inf.uni-due.de

Abstract

In this paper, we present YalAA, a new library for affine arithmetic. Recently, affine arithmetic has been given increased attention even from outside the traditional verified computing community, for example, in the areas of circuit design, GPU based rendering of implicit objects and global optimization. Furthermore, several improvements to the original affine model were proposed. However, a fully verified, object-oriented implementation supporting at least some of the extensions is currently not available.

The goal of YalAA is to provide a wide range of elementary functions and to allow the user to incorporate improvements for the original affine model easily. In contrast to other available libraries, YalAA also provides verified implementations of non-convex or non-concave elementary functions. Our library has a policy based design. That is, the user can alter different aspects in the library's behavior to reflect variations of the original model while relying on the same code base. Because affine arithmetic is often used in combination with interval arithmetic, we followed the principles of the upcoming P1788 interval standard during the design process. Therefore, YalAA can be integrated seamlessly into an existing interval arithmetic environment.

Keywords: affine arithmetic, library, interval analysis, yalaa, policy based design
AMS subject classifications: 65-04

1 Introduction

Common interval arithmetic (IA) [1] can provide a verified range enclosure for functions constructed of basic operations and elementary functions. However, these enclosures are not always tight and can overestimate the true range. This is mainly due to the *dependency problem* and *wrapping effect*. IA treats every occurrence of a variable as independent during the computation process, resulting in the former. The latter is

[*]Submitted: September 12, 2011; Revised: September 8, 2012; Accepted: September 10, 2012.

caused by the fact that every interval is axis-aligned. Geometrically speaking, if an interval box is rotated, for example, by 45 degrees, the result has to be wrapped into an axis-aligned box, increasing the area of the original box by a factor of $\sqrt{2}$.

Overestimation can yield unusable results due to their wide range. Furthermore, it can increase the computation times in the often used interval branch-and-bound algorithms significantly. Several more sophisticated methods for verified range bounding were proposed in order to reduce such overestimation. Among these are such techniques as mean-value forms [22], Taylor models [2], ellipsoid arithmetic [23], generalized interval arithmetic (GIA) [10] or affine arithmetic (AA) [5].

GIA, proposed by E. Hansen in 1975, tracks the result's first order dependency on the input variables. AA is a very similar model presented in 1990 by Comba and Stolfi. In contrast to GIA, it does not only track the dependency on the input variables but also introduces variables for errors occurring during the computation. Furthermore, it uses real values instead of intervals for storing the dependencies. Originally, AA was promoted as a verified arithmetic tailored for computer graphics. Today it is applied to a wide range of problems such as GPU based raytracing of implicit surfaces [16], [8], circuit design [18] or global optimization [25], [20].

2 Affine Arithmetic

In this section, we give a short overview of AA. Having introduced the basic model, we discuss some of the recently suggested improvements for it. After that, we describe the two publicly available implementations LIBAA and LIBAFFA.

2.1 Basic Model

In AA, a partially unknown quantity $\hat{x}$ is represented as an affine combination of a central value x_0 and error terms $x_i\epsilon_i$

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\epsilon_i. \tag{1}$$

The x_i are real numbers[1] and called partial deviations, whereas the ϵ_i are symbolic noise variables. They model linear dependencies between affine forms and are assumed to lie inside the interval $[-1,1]$. If two affine forms share the same noise symbol ϵ_i, there is a partial linear dependency between them. The joint range of affine forms is visualized as a center symmetric polytope.

We can express affine operations (that is, addition, scaling and translation) naturally inside the AA model:

$$\begin{aligned}
\hat{x} \pm \hat{y} &= (x_0 \pm y_0) + (x_1 \pm y_1)\epsilon_1 + \ldots + (x_n \pm y_n)\epsilon_n \ , \\
\alpha\hat{x} &= (\alpha x_0) + (\alpha x_1)\epsilon_1 + \ldots + (\alpha x_n)\epsilon_n \ , \\
\alpha \pm \hat{x} &= (x_0 \pm \alpha) + x_1\epsilon_1 + \ldots + x_n\epsilon_n \ .
\end{aligned}$$

A non-affine function $f(\hat{x})$ cannot be expressed directly in terms of AA. Instead we have to choose an affine approximation $f^a(\hat{x})$, with which we calculate the affine part of f and enclose the approximation error. The error is added as a new term $x_{n+1}\epsilon_{n+1}$ to the resulting affine form. Here, ϵ_{n+1} should be a previously unused noise

[1] or floating-point numbers on a computer

variable. Rounding errors, occurring irrespectivly of whether f is affine or non-affine, are handled similarly. In the non-affine case, the rounding errors are usually added to the approximation error.

2.2 Improvements to the Original Model

We can group most AA improvements into three categories. The first category contains all suggestions which do no alter the model itself but propose a different implementation technique. An example is the talk [24], which describes an approach for rounding control differing from the original implementation [6], or the paper [17]. In the latter, the authors employ a better routine for the multiplication of affine forms. The second category consists of improvements which alter the model but still produce an affine form. For example the AF1 and AF2 forms are introduced in [20][2]. They feature a new way of handling approximation errors. Changes creating a non-affine model belong in the third and last category. For example, the authors in [21], [4] and [15] propose to preserve higher order noise symbols ϵ_i^n to keep track of higher order dependencies.

We restrict the further discussion to affine models because YalAA only support those at present. In AA, the number of error terms in an affine form increases during the computation process. Usually, one extra error term is added with each basic operation or elementary function call. None of these extra terms model any *affine* dependency on the input variables[3]. Therefore, they often do not offer any advantage but increase the computational cost. The AF1 form changes the original model by introducing a *special* error term $x_{n+1}\epsilon_{n+1}$ where all the uncertainty introduced into the computation process by approximation and rounding errors is stored.

The AF2 model extends AF1 by splitting the special error term $x_{n+1}\epsilon_{n+1}$ into three terms $x_{n+1}\epsilon_{n+1}, x^+_{n+1}\epsilon^+_{n+1}$ and $x^-_{n+1}\epsilon^-_{n+1}$ describing general, positively signed and negatively signed errors, respectively. Signed errors usually occur during multiplication and integer power operations, as these produce noise variables of the form $\epsilon_i^{2k}, k \in \mathbb{N}$. Consider, for example, the square of $\hat{x} = x_0 + x_1\epsilon_1$ resulting in $(\hat{x})^2 = x_0^2 + 2x_0x_1\epsilon_1 + x_1^2\epsilon_1^2$. The term $x_1^2\epsilon_1^2$ is nonlinear, thus we have to approximate it by a new noise symbol. However, we know that $\epsilon_1^2 \in [0,1]$, that is, the approximation error is positively signed.

In [28], the authors introduced a variant of AA which they call revised affine arithmetic (RevAA). An affine form in RevAA consists of an affine part and an interval part enclosing the nonlinear noise $\hat{x} = x_0 + x_1\epsilon_1 + \ldots + x_n\epsilon_n + e_x[-1,1]$. Therefore, a computation between two revised forms consists of two operations: The operation on the affine part and the interval computation. Similar to the forms AF1/AF2 introduced by Messine, the length of the affine part can not grow larger than the number of input variables.

The publicly available implementations of AA take into account rounding errors as described in [6]. That is, the exact floating-point error is computed during an affine operation by performing the same calculation with three rounding modes: to nearest, to $+\infty$, to $-\infty$. In [24], Ninin et al. propose a new method for calculating the rounding error. The authors suggest using small interval coefficients and letting IA take care of the rounding errors, which makes the model very similar to GIA. The second method proposed there is an a posteriori error correction similar to that used

[2]There is a minor mistake in the multiplication routine for two AF2 forms presented in this paper, see [28] for a corrected version.

[3]They depend on the input nonlinearly.

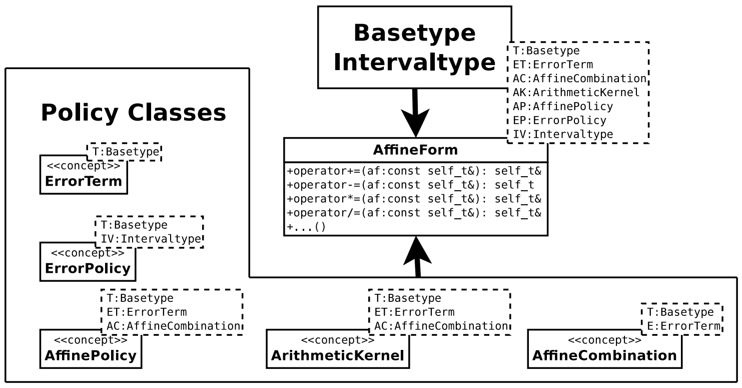

Figure 1: Basic structure of YalAA's design.

for Taylor models in COSY [3], [26].

2.3 Existing Implementations

Currently, there are two publicly available implementations of the AA intended for verified computations: LIBAA [27] and LIBAFFA [9]. The former is the reference implementation created by the inventors of AA. It is written in C and is based on a non-object-oriented stack approach. It supports only a very limited set of elementary functions not including non-convex or non-concave functions like the sine and the cosine. LIBAFFA is implemented in C++, provides an object-oriented interface and at least the basic trigonometric functions. However, the derivation of the approximation error and the handling of rounding errors is not always correct. Both libraries follow the model description and implementation outlined in [6]. Hence, they only implement the original affine model.

The goal of YalAA is to provide a fully verified, object-oriented AA implementation with a complete set of elementary functions. Furthermore, a user should be able to modify YalAA's behavior easily in order to implement extended models without rewriting major parts of the library. The set of supported elementary functions in YalAA is based on the set required by the upcoming P1788 interval standard [7]. To provide the intended flexibility, we chose a policy based design. That is, the behavior can be altered by one small policy class, which does not affect other parts of the library.

3 Design of YalAA

Basically, YalAA is constructed of one main class and five policy classes as shown in Fig. 1. The library can be customized further with the two types `T` and `IV`. The former is used for representing the partial deviations and the central value. It is called the base type and allows users to customize types for deviations. For example, they can employ exact rational numbers instead of floating-point numbers. The underlying interval type is specified by `IV`. As YalAA uses a trait class for accessing it, any existing interval library can be integrated seamlessly. Furthermore, users can mix affine forms with the

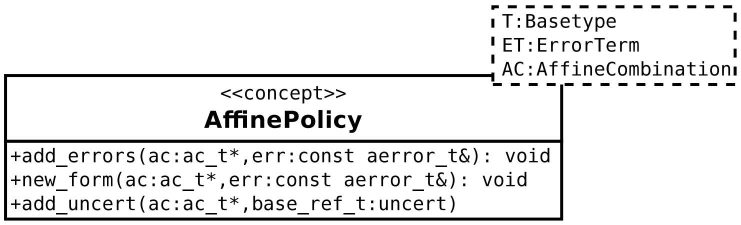
T:Basetype
ET:ErrorTerm
AC:AffineCombination
<<concept>>
AffinePolicy
+add_errors(ac:ac_t*,err:const aerror_t&): void
+new_form(ac:ac_t*,err:const aerror_t&): void
+add_uncert(ac:ac_t*,base_ref_t:uncert)

Figure 2: The `AffinePolicy` class in detail.

types `T` and `IV` in the basic operations. This is important because AA is often employed inside existing IA environments. In the remainder of this section, we describe each of the classes in Fig. 1 in detail.

ErrorTerm The policy class `ErrorTerm` represents the i-th error term $x_i\epsilon_i$. It provides an ordering on the error terms and is responsible for generating new noise variables. The latter operation has to be thread-safe if the library is used in a multi-threaded environment. Furthermore, it stores the unique identifier for noise symbols and thus limits their maximum number. Such operations are possibly performance critical. Therefore, users can adapt them to their current needs by changing the policy.

AffineCombination The `AffineCombination` policy class is responsible for storing an affine combination of the central value and the error terms. Further, it provides the affine operations addition, scaling and translation. Any operation performed in YalAA is broken down into these. Note that no rounding control, error correction or handling is done at this particular level, only the plain basic operations.

ArithmeticKernel The core component of YalAA is the `ArithmeticKernel` policy. It provides an implementation of all supported operations and elementary functions and performs the affine part of them. Furthermore, it calculates bounds on the approximation and rounding error and tracks exceptional situations occurring during the computation, for example domain violations or overflows. This information is propagated through the `ArithmeticError` class which acts as a uniform interface between possibly different implementations of `ArithmeticKernel` and the rest of YalAA. As the verified implementation of an operation depends heavily on the underlying type, `ArithmeticKernel` has to be specialized for each base type `T`.

AffinePolicy The `AffinePolicy` class shown in Fig. 2 is responsible for handling rounding and approximation errors, creating new affine forms and introducing uncertainty into the computation. The three methods look similar, but have different semantics. The method `add_errors` is called for adding a particular way of handling rounding and approximation errors to affine combinations. `New_form` is called if a new affine form is created, that is, a new *input* variable is introduced. `Add_uncert` is called if uncertainty is introduced into the computation process, for example, an affine form is combined with an interval. Utilizing the three methods, we can implement the usual

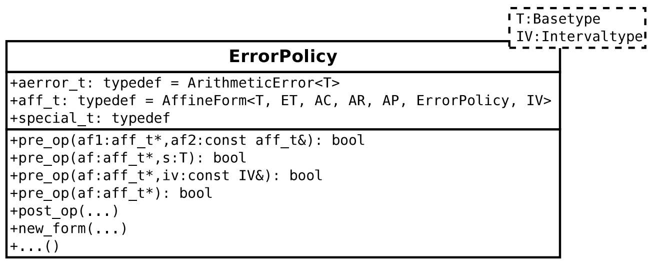
T:Basetype
IV:Intervaltype
ErrorPolicy
+aerror_t: typedef = ArithmeticError<T>
+aff_t: typedef = AffineForm<T, ET, AC, AR, AP, ErrorPolicy, IV>
+special_t: typedef
+pre_op(af1:aff_t*,af2:const aff_t&): bool
+pre_op(af:aff_t*,s:T): bool
+pre_op(af:aff_t*,iv:const IV&): bool
+pre_op(af:aff_t*): bool
+post_op(...)
+new_form(...)
+...()

Figure 3: The `ErrorPolicy` in detail.

Table 1: Supported state propagation flags in YalAA.

Flag	Description
`DISCONT`	Function is discontinuous over the arguments.
`UNBOUND`	Function is not bounded over the arguments.
`P_D_VIOL`	Function is partially undefined over the arguments.
`C_D_VIOL`	Function is undefined over the arguments.
`OFLOW`	An overflow occurred.
`I_ERROR`	An internal error occurred.

affine model and the extended AF1 and AF2 models just by exchanging the policy class[4].

ErrorPolicy The `ErrorPolicy` shown in Fig. 3 is responsible for providing special value types `special_t` for affine forms, for example, the empty set or the whole real line, and for handling information about exceptional situations passed on by the `ArithmeticKernel`. YalAA supports propagation of mutual combinations of the six states shown in Tab. 1.

The `pre_op` methods are called prior to an operation. They can prevent YalAA from actually performing the operation, if the result is always a special value in the current error policy. The `post_op` method is called after the actual operation and handles exceptional situations occurring during the operation. With the `ErrorPolicy` approach, we can implement different concepts for error handling, for example, the common error handling for AA described in [6] or even a decoration-like approach as currently discussed for intervals in the P1788 standardization group.

The complete process for evaluating an elementary function or a basic operation on an affine form is outlined in Fig. 4. In the first step, the user calls an operation. Then a check for special affine forms in the input arguments is performed by `ErrorPolicy::pre_op`. If none are present, the actual operation is called in the `ArithmeticKernel` class. The `AffinePolicy` adds the rounding and approximation errors. In the last step, `ErrorPolicy::post_op` checks for errors which possibly occured during the operation.

[4]The special noise symbols introduced by AF1/AF2 also need support at the level of the basic operations, which the `AffineCombination` policy class provides.

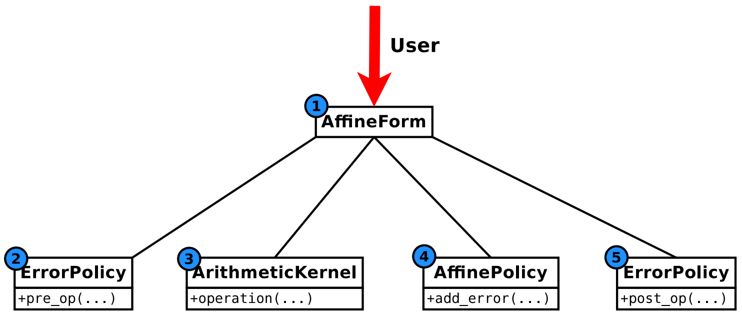

Figure 4: Interaction of YalAA's policy classes in a function call.

4 Policy Classes

In the following section we describe the policy classes supplied with YalAA. These classes let users either to employ the library directly or to customize it according to their needs.

4.1 Affine Policies

Currently, we supply three implementations of the `AffinePolicy` in YalAA. Following [20], we name them AF0, AF1 and AF2. AF0 is the usual affine computation model, whereas AF1 and AF2 implement the respective extended models. In the following, we describe these three policies in detail.

AF0 All operations in `AffinePolicy` introduce a new noise variable ϵ_{n+1}. Signed errors are handled by scaling the central value. If e^+ is a positively signed error, then the scaled central value and the respective deviation are computed according to

$$\begin{aligned} x_0 &= x_0 + 0.5e^+ \ , \\ x_{n+1} &= 0.5e^+ \ . \end{aligned}$$

AF1 In contrast to AF0, the AF1 policy maps all errors into one single special error term. Only the `new_form` method in `AffinePolicy` introduces a new noise symbol in this model. The two other methods add the error/uncertainty to the special error term. Signed errors are handled as in AF0.

AF2 AF2 is very similar to AF1 but splits the special error term. Therefore it is necessary to alter the handling of signed errors. The policy adds them to their respective special error terms $x_{n+1}\epsilon_{n+1}, x_{n+1}\epsilon^+_{n+1}$ and $x_{n+1}\epsilon^-_{n+1}$.

4.2 Error Policies

The `ErrorPolicy` classes allow the user to customize the behavior of the library in case of domain violations, overflows or internal errors. Currently, YalAA provides policies implementing the error handling technique used in `libaffa` and `libaa` and a decoration-like model. In the following subsection, we describe the implementation of this two policies in detail.

Table 2: Combinations of special forms in the common error model.

$\circ$	NONE	R	EMPTY
NONE	NONE	R	EMPTY
R	R	R	EMPTY
EMPTY	EMPTY	EMPTY	EMPTY

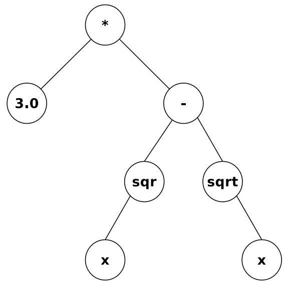

Figure 5: Computational graph for the inductively defined function $3.0(x^2-\sqrt{x})$.

4.2.1 Common Error Model

The standard model for error handling in AA was introduced in [6] and is used in LIBAA and LIBAFFA. It employs two special affine forms `R` and $\emptyset$. The former represents the whole real line and indicates an unbounded or overflowed result. The latter is the empty set $\emptyset$ and used if the operation is not defined. Table 2 shows the outcomes of combinations of special values. They are used by the `ErrorPolStd`, which implements this model in YALAA, for the `pre_op` operation.

4.2.2 Decoration Model in YalAA

A more sophisticated approach to error handling called decorations is currently discussed by the P1788 interval standardization group [13], [11], [12]. The key idea is to propagate a decoration through the computational graph of an inductively defined function. A function is inductively defined if it is a composition of a finite number of basic operations and elementary functions like that shown in Fig. 5. In particular, any function calculated using the natural affine extension[5] is inductively defined. That is, each function that can be directly computed using YALAA fulfills this requirement.

The P1788 decoration support is still work in progress and is tailored towards IA. We use a similar but not completely identical mechanism. In YALAA, decorations are supplied by the `ErrorPolDec` policy and are either features of the function or error states. YALAA tracks three attributes through decorations: "domain tetrit" D, "defined and continuous" C and "defined and bounded" B.

Let $D_f \subseteq \mathbb{R}^n$ be the natural domain of $f : \mathbb{R}^n \to \mathbb{R}$. Then the domain tetrit

[5]If all real operations, quantities and elementary functions in an expression are replaced by their respective AA counterparts, we obtain the natural affine extension.

Table 3: Decorations supported by YalAA.

D^+	D^-	C	B	Dec.	Meaning
T	F	T	T	$\mathbb{D}_5$	f is cert. defined, cont. and bounded over $\boldsymbol{x}$
T	F	T	F	$\mathbb{D}_4$	f is cert. defined and cont. over $\boldsymbol{x}$
T	F	F	F	$\mathbb{D}_3$	f is cert. defined over $\boldsymbol{x}$
T	T	F	F	$\mathbb{D}_2$	f is possibly defined over $\boldsymbol{x}$
F	T	F	F	$\mathbb{D}_1$	f is certainly undefined over $\boldsymbol{x}$
F	F	F	F	$\mathbb{D}_0$	$\boldsymbol{x}$ is the empty set
?	?	?	?	$\mathbb{D}_{-1}$	an error occurred

Table 4: Connection between affine part and decoration.

Decoration	Meaning for the affine part	Central value (IEEE754 types)
$\mathbb{D}_5$	Has a result	*valid*
$\mathbb{D}_4$	Overflowed	$\pm\infty$
$\mathbb{D}_3$	Possibly has a result	*valid* or $\pm\infty$
$\mathbb{D}_2$	Possibly has a result	*valid* or $\pm\infty$
$\mathbb{D}_1$	Is the empty set	NaN
$\mathbb{D}_0$	Is the empty set	NaN
$\mathbb{D}_{-1}$	Undefined	NaN

$D(f, \boldsymbol{x}) = (D^+, D^-)$ for an interval $\boldsymbol{x} \in \mathbb{IR}^n$ is defined as

$$D^+ \Leftrightarrow (\exists x \in \boldsymbol{x}) : \boldsymbol{x} \in D_f$$
$$D^- \Leftrightarrow (\exists x \in \boldsymbol{x}) : \neg(\boldsymbol{x} \in D_f).$$

The two other flags are defined as:

$$C(f, \boldsymbol{x}) \Leftrightarrow \text{ The restriction of f to } \boldsymbol{x} \text{ is defined and continuous.}$$
$$B(f, \boldsymbol{x}) \Leftrightarrow \text{ The restriction of f to } \boldsymbol{x} \text{ is defined and bounded.}$$

The possible combinations of these attributes and the resulting decorations are listed in Tab. 3. Each decoration except $\mathbb{D}_{-1}$ and $\mathbb{D}_0$ describe properties of the function. They are retrieved for the interval box enclosing the affine form and ordered, that is, $\mathbb{D}_i < \mathbb{D}_{i-1}$ for $5 \geq i \geq 0$. A decoration of a function $f(\hat{x})$ is the minimum decoration of its arguments and the function's decorations over $[a, b]$ enclosing $\hat{x}$.

Based on the result's decoration, the user can deduce whether the affine part has a meaningful value as shown in Tab. 4. The current approach to decorations is not ideal as it is unclear from the decorations $\mathbb{D}_3$ and $\mathbb{D}_2$ whether the affine part is valid.

YalAA allows the direct composition of affine forms with intervals or scalars. As currently neither of them have a decoration, we either have to force the user to perform an explicit conversion to a decorated affine form or to provide some automatic mechanism. The rules for automatic conversion are given in Tab. 5. In short, we assign to an interval or a scalar the best decoration $\mathbb{D}_5$ if it is not a special value or empty. In IEEE754, special values are NaN or $\pm\infty$. Although assigning the best decoration automatically is disputable, in our experience the intervals or scalars used for composition are mostly constants and as such can be considered *safe*. If not, the user still has the opportunity to perform a manual conversion.

Table 5: Automatic conversion rules for intervals and scalars in composition with decorated affine forms.

Type	**Value**	**Dec.**
Scalar	$\neg$`special`	$\mathbb{D}_5$
Scalar	`special`	$\mathbb{D}_{-1}$
Interval	$\neg$(`empty` $\vee$ `special`)	$\mathbb{D}_5$
Interval	`empty` $\wedge\neg$`special`	$\mathbb{D}_0$
Interval	`special`	$\mathbb{D}_{-1}$

5 Floating-Point Implementation

YalAA is supplied with a specialization of the `ArithmeticKernel` class for the floating-point types `float`, `double` and `long double`. Where suitable, YalAA works with the algorithms described in [6]. In particular, it uses the exact calculation of the rounding error for affine functions and min-range approximation for non-affine elementary functions. However, the latter is only applicable to convex or concave functions. We handle other elementary functions by a Chebyshev interpolation based approach.

For constructing verified affine approximations of elementary functions, both min-range and Chebyshev interpolation need lower and upper bounds on the elementary functions. As the standard math libraries provided by the C++ environment neglect user-defined rounding modes, YalAA uses IA for evaluating all elementary functions except the square root during the approximation process.

5.1 Chebyshev Interpolation

For computing a non-affine function f in AA, we need an affine approximation which can be derived on the basis of the Chebyshev interpolation. Following [6], we limit our discussion to univariate affine approximations of the form $\alpha x + \zeta$. The Chebyshev nodes x_k are defined as

$$x_k = \cos\left(\frac{\pi(2k+1)}{2n+2}\right), k = 0\dots n,$$

and are the roots of the Chebyshev polynomials

$$T_i(x) = \cos i\theta \text{ , if } x = \cos\theta \text{ ,}$$

see [19]. A function $f : [-1,1] \to \mathbb{R}$ can be approximated using the n-th degree Chebyshev interpolant

$$p_n(x) = \frac{c_0}{2} + \sum_{k=1}^{n} c_k T_k(x)$$

with the Chebyshev coefficients

$$c_i = \frac{2}{n+1}\sum_{k=0}^{n} f(x_k) T_i(x_k).$$

For approximating a function on a general finite interval $[a,b]$, a linear transformation to $[-1,1]$ is necessary. The new Chebyshev nodes are obtained through the inverse transformation as

$$x'_k = \frac{1}{2}\left((b-a)x_k + a + b\right)$$

so that the coefficients are equal to

$$c'_i = \frac{2}{n+1} \sum_{k=0}^{n} f(x'_k) T_i(x_k).$$

The new interpolant is given as

$$p_n(x) = \frac{c_0}{2} + \sum_{k=0}^{n} c'_k T_k(x)$$

for $x \in [-1, 1]$. We can transform any $x' \in [a, b]$ using a linear transformation $t : [a, b] \to [-1, 1]$ with

$$t(x') = \left(\frac{2x' - (a+b)}{b-a} \right).$$

Therefore, the final polynomial for x' is

$$p_n(x') = \frac{c_0}{2} + \sum_{k=0}^{n} c'_k T_k \left(\frac{2x' - (a+b)}{b-a} \right).$$

We want to compute an affine approximation of the form $\alpha\hat{x} + \zeta \pm \delta$ over the domain $\boldsymbol{x} = [a, b]$ of $\hat{x}$. This is a polynomial of degree one, so that we have to compute only the coefficients c'_0 and c'_1. We calculate enclosures

$$\boldsymbol{\alpha} = \frac{2c'_1}{b-a}$$

and

$$\boldsymbol{\zeta} = \frac{c'_0}{2} - \frac{a+b}{b-a}$$

for α and ζ in IA. We take the midpoints of $\boldsymbol{\alpha}, \boldsymbol{\zeta}$ and shift the rounding error into δ according to

$$\delta = \frac{1}{2} \left(\text{len}(\hat{x}) \text{width}\, \boldsymbol{\alpha} + \text{width}\, \boldsymbol{\zeta} \right).$$

Here, $\text{len}(\hat{x})$ denotes the number of noise symbols in $\hat{x}$. For rigorous results, we have to use rounding towards $+\infty$ for δ. A bound for the approximation errors can be derived with Lagrange's remainder formula:

$$R(\boldsymbol{x}) = \frac{(\text{width}\, \boldsymbol{x})^2 f^{(2)}(\boldsymbol{x})}{16},$$

if the second derivative is available.

As the central value x_0 is always the midpoint of the interval $[a, b]$ enclosing the affine form, the linear transformation t will always map x_0 to zero in exact arithmetic. Therefore, instead of the direct affine transformation $\hat{y} = \alpha\hat{x} + \zeta$ we use the following formula:

$$\begin{aligned} y_0 &= \zeta \\ y_i &= \alpha x'_i \end{aligned}$$

In order to bound $R(\boldsymbol{x})$, we usually use the natural interval extension of $f^{(2)}(\boldsymbol{x})$. Because of the dependency effect, we sometimes have to exploit other features of $f^{(2)}$, like monotonicity, in order to get sharp bounds on $R(\boldsymbol{x})$ (e.g. for the inverse tangent). Another problem occurs if a function is not two times differentiable over its domain. In this case, we cannot use the Lagrange remainder formula. In our library, this is only the case for the inverse sine and cosine functions. Both functions have the domain $[-1,1]$, their respective second derivatives $\pm\frac{x}{\left(1-x^2\right)^{3/2}}$ are undefined at the endpoints. For both functions, we calculate

$$e(x) = |f(x) - p_1(x)|_\infty = \max_{\forall x \in [a,b]} |f(x) - p_1(x)|,$$

that is the exact error of our approximation. The maximum error can occur either at the endpoints of $[a,b]$ or at a local maximum. The local maximum is x^* such that $f'(x^*) - p_1'(x^*) = 0$. If we solve this equation we get $x_{1|2} = \pm\sqrt{1-\frac{1}{c_1^2}}$ as candidates for the maximum. We have to evaluate $e(x)$ at all 4 points a, b, x_1, x_2 with IA in order to derive a verified upper bound. This technique is much more expensive than the ordinary Lagrange remainder, hence we only apply it near the endpoints, where the derivatives behave poorly.

5.2 Integer Power Function

In our implementation, the integer power function $\texttt{pown}(\hat{x}, n)$ is a direct extension of the squaring approach used in GIA:

$$\begin{aligned}
\hat{x}^n &= \left(x_0 + \sum_{i=1}^m x_i\epsilon_i\right)^n \\
&= x_0^n + nx_0^{n-1}\left(\sum_{i=1}^m x_i\epsilon_i\right) + \\
&\binom{n}{2}x_0^{n-2}\left(\sum_{i=1}^m x_i\epsilon_i\right)^2 + \binom{n}{3}x_0^{n-3}\left(\sum_{i=1}^m x_i\epsilon_i\right)^3 + \ldots + \left(\sum_{i=1}^m x_i\epsilon_i\right)^n .
\end{aligned}$$

The first two terms of the sum are the affine part of the power function. The nonlinear terms are enclosed by a new error term. If we denote the radius of $\hat{x}$ by

$$\operatorname{rad}\hat{x} = \sum_{i=1}^m |x_i| \ ,$$

we can bound the error as

$$e = \sum_{k=2}^n \binom{n}{k} (\operatorname{rad}\hat{x})^k.$$

We split the error into the unsigned part e^+ for all terms with an even k and the signed part $e^\pm$ for all odd terms.

6 Comparison: YalAA vs. Other Implementations

In Tab. 6, we list the elementary functions required in the upcoming interval standard and compare their support status in AA libraries. According to P1788, the power function `pow` has to be the *general power function.* Neither LIBAFFA nor YALAA support it. Instead, both use the usual definition $x^y = \exp^{y \ln x}$. Additionally, YALAA provides the rational power function $x^{\frac{p}{q}}$ and the n-th root $x^{\frac{1}{p}}$, $p \in \mathbb{Z}, q \in \mathbb{N}$ to complement `pow`. Both LIBAFFA and YALAA support a wide range of elementary functions. However, the implementation for at least the sine and cosine functions is

Table 6: Support of P1788 required elementary functions in AA libraries.

Function	libaa	libaffa	YalAA
sqr	✓	✓	✓
pown	×	✓	✓
pow	×	✓	✓
sqrt	✓	✓	✓
exp	✓	✓	✓
exp2, exp10	×	×	✓
log	×	✓	✓
log2, log10	×	×	✓
expm1, exp2m1, exp10m1	×	×	✓
logp1, lo2p1, log10p1	×	×	✓
sin	×	✓	✓
cos	×	✓	✓
tan	×	✓	✓
asin	×	×	✓
acos	×	×	✓
atan	×	×	✓
atan2	×	×	×
sinh	×	✓	✓
cosh	×	✓	✓
tanh	×	✓	✓
asinh	×	✓	✓
acosh	×	✓	✓
atanh	×	✓	✓
abs	✓	✓	×
rSqrt	×	×	×
hypot	×	×	×
compoundm1	×	×	×

not verified in LIBAFFA as it does not derive verified bounds on the remainder. Also, it uses directed rounding with floating-point elementary functions, which does not guarantee correct results.

All libraries implement the usual error model for exception handling. In addition, YALAA provides a limited support for decorations. Furthermore, YALAA can be used with arbitrary base types in contrast to the other libraries, which rely on fixed data types for representing affine forms. YALAA can use external interval libraries through trait classes, for example, C-XSC [14] or filib++. Therefore, it can easily interact with them whereas LIBAFFA has a built-in IA library and LIBAA always employs a fixed external library.

7 Conclusion and Outlook

In this paper, we presented YALAA, a new library for affine arithmetic. To our knowledge, YALAA provides the first verified publicly available implementation of non-convex or non-concave elementary functions in affine arithmetic along with a wide range of P1788 recommended elementary functions. Its policy based design allows easy adaption to the users' needs. In contrast to existing libraries, YALAA can be integrated seamlessly into existing interval environments allowing for mixed operations with intervals.

In the future, we plan to add support for all elementary functions required or recommended by P1788 standard. Another goal is to implement an a posteriori error correction such as the one suggested in [24]. As the GIA and the AA computation models are very similar, it is also interesting to support GIA inside YALAA. This would make a fair comparison of these two computation models possible.

References

[1] G. Alefeld and J. Herzberger. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[2] M. Berz. Modern map methods for charged particle optics. *Nuclear Instruments and Methods A363*, pages 100–104, 1995.

[3] M. Berz and K. Makino. Cosy infinity 9.0. Technical Report MSUHEP 060803, Michigan State University, 2006.

[4] G. Bilotta. Self-verified extension of affine arithmetic to arbitrary order. *Le Matematiche*, 63(1):15–30, 2008.

[5] J.L.D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. *Presented at SIBGRAPI'93, Recife, PE (Brazil)*, 1993.

[6] L.H. de Figueiredo and J. Stolfi. *Self-Validated Numerical Methods and Applications.* IMPA, Rio de Janeiro, 1997.

[7] J. Pryce (Tech. Eds.). P1788: IEEE Standard for Interval Arithmetic Version 02.2.

[8] Oleg Fryazinov, Alexander Pasko, and Peter Comninos. Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic. *Computers & Graphics*, 34(6):708 – 718, 2010.

[9] O. Gay, D. Coeurjolly, and N.J. Hurst. libaffa. `http://www.nongnu.org/libaffa/`, accessed on 06.09.2012.

[10] E. Hansen. A generalized interval arithmetic. In Karl Nickel, editor, *Interval Mathematics*, volume 29 of *Lecture Notes in Computer Science*, pages 7–18. Springer Berlin / Heidelberg, 1975.

[11] N. T. Hayes. Trits to tretrits, 2010. P1788, Motion 18.

[12] N. T. Hayes. Property tracking with decorations, May 2011. P1788, Proposed Motion.

[13] N. T. Hayes and A. Neumaier. Exception handling for interval arithmetic. P1788, Motion 8.

[14] W. Hofschuster and W. Krämer. C-XSC 2.0 A C++ library for extended scientific computing. In Ren Alt, Andreas Frommer, R. Kearfott, and Wolfram Luther, editors, *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 259–276. Springer Berlin / Heidelberg, 2004.

[15] S. Huahao, L. Hongwei, M. Ralph, and W. Guojin. Modified affine arithmetic is more accurate than centered interval arithmetic or affine arithmetic. In Michael Wilson and Ralph Martin, editors, *Mathematics of Surfaces*, volume 2768 of *Lecture Notes in Computer Science*, pages 355–365. Springer Berlin / Heidelberg, 2003.

[16] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen. Fast ray tracing of arbitrary implicit surfaces with interval and affine arithmetic. *Computer Graphics Forum*, 28(1):26–40, 2009.

[17] V. L. Kolev. Optimal multiplication of g-intervals. *Reliable Computing*, 13:399–408, 2007.

[18] A. Lemke, L. Hedrich, and E. Barke. Analog circuit sizing based on formal methods using affine arithmetic. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 486–489. ACM, 2002.

[19] J. C Mason and D. C. Handscomb. *Chebyshev Polynomials.* CRC Press, 2003.

[20] F. Messine. Extentions of affine arithmetic: Application to unconstrained global optimization. *Journal of Universal Computer Science*, 8(11):992–1015, 2002.

[21] F. Messine and A. Touhami. A general reliable quadratic form: An extension of affine arithmetic. *Reliable Computing*, 12:171–192, 2006.

[22] A. Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[23] A. Neumaier. The wrapping effect, ellipsoid arithmetic, stability and confidence regions. *Computing (Suppl.)*, 9:175–190, 1993.

[24] J. Ninin and F. Messine. Reliable affine arithmetic, 2011. International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2010).

[25] J. Ninin, F. Messine, and P. Hansen. A reliable affine relaxation method for global optimization. Technical report, IRIT, 2010.

[26] N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in cosy. *Journal of Logic and Algebraic Programming*, 64(1):135–154, 2005.

[27] J. Stolfi. libaa. `http://www.ic.unicamp.br/~stolfi/`, accessed on 06.09.2012.

[28] Xuan-Ha Vu, Djamila Sam-Haroud, and Boi Faltings. Enhancing numerical constraint propagation using multiple inclusion representations. *Annals of Mathematics and Artificial Intelligence*, 55:295–354, 2009.