Implementation of affine arithmetic using floating-point arithmetic: how to handle roundoff errors

Jordan Ninin
IHSEV team, LAB-STICC, ENSTA-Bretagne, France
Nathalie Revol
INRIA - Université de Lyon - LIP, ENS de Lyon, France

SCAN 2014, Würzburg, Germany, 22 September 2014

affine arithmetic and applications

**computer arithmetic
floating-point arithmetic
roundoff errors**

Question: **Is the inclusion property preserved?**
Question: **Are roundoff errors accounted for?**

**implementation of affine arithmetic
using floating-point arithmetic:
fast and accurate**

affine arithmetic
and applications

computer arithmetic
floating-point arithmetic
roundoff errors

Question: **Is the inclusion property preserved?**
Question: **Are roundoff errors accounted for?**

implementation of affine arithmetic
using floating-point arithmetic:
fast and accurate

affine arithmetic
and applications

computer arithmetic
floating-point arithmetic
roundoff errors

Question: **Is the inclusion property preserved?**
Question: **Are roundoff errors accounted for?**

implementation of affine arithmetic
using floating-point arithmetic:
fast and accurate

affine arithmetic
and applications

computer arithmetic
floating-point arithmetic
roundoff errors

Question: **Is the inclusion property preserved?**
Question: **Are roundoff errors accounted for?**

implementation of affine arithmetic
using floating-point arithmetic:
fast and accurate

Contributions

Roundoff errors:

bounded computed

Use of directed roundings:

yes no

Influence of the architecture?

Agenda

Agenda

Affine arithmetic

Comba, de Figueiredo, Stolfi – Vu, Sam-Haroud, Faltings

Variant of interval arithmetic, thus guaranteed enclosures of the sought result.

Why? To counteract the dependency problem.
With affine arithmetic, $\mathbf{x} - \mathbf{x} = [0]$.

How? Each quantity is a linear combination of noise symbols:

$$\hat{\mathbf{x}} = x_0 + \sum_i x_i \epsilon_i \text{ where } x_i \in \mathbb{R} \text{ and } \epsilon_i \text{ lives in } [-1, 1].$$

Affine arithmetic: operations

Comba, de Figueiredo, Stolfi

Notations

$$\alpha \in \mathbb{R}, \beta \in \mathbb{R},$$

$$\hat{\mathbf{x}} = x_0 + \sum_i x_i \epsilon_i \text{ where } x_i \in \mathbb{R} \text{ and } \epsilon_i \text{ lives in } [-1, 1],$$

$$\hat{\mathbf{y}} = y_0 + \sum_i y_i \epsilon_i \text{ where } y_i \in \mathbb{R} \text{ and } \epsilon_i \text{ lives in } [-1, 1].$$

$$\alpha + \beta \hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i) \epsilon_i.$$

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = \ldots$$

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_{\pm}$.

Variations exist...

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_{\pm}$.

Variations exist...

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_{\pm}$.

Variations exist...

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_\pm$.

Variations exist...

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_\pm$.

Variations exist...

Affine arithmetic: multiplication... and limiting the number of noise symbols

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = x_0 \times y_0 + \ldots + \sum_i (x_0 y_i + x_i y_0)\epsilon_i + \epsilon_{n+1}$$

ϵ_{n+1} is a new symbol created to account for nonlinear terms.

Explosion of the number of symbols: handled by fixing the maximal number of such symbols.

Handling the non-created symbols? through a dedicated symbol $\epsilon_\pm$.

Variations exist...

Affine arithmetic: handling of roundoff errors

Implementation using floating-point arithmetic.

Question: **Is the inclusion property preserved?**

Question: **Are roundoff errors accounted for?**

Affine arithmetic: handling of roundoff errors

Comba, de Figueiredo, Stolfi

$$\alpha + \beta\hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error. Ex.: $\beta \times x_i$.

Roundoff error computed as:
$e = \max(\mathrm{RU}(\beta \times x_i) - \mathrm{RN}(\beta \times x_i), \mathrm{RN}(\beta \times x_i) - \mathrm{RD}(\beta \times x_i))$.

Roundoff errors accumulated in $\epsilon_\pm$:

$$\epsilon_\pm = \mathrm{RU}(\epsilon_\pm + e).$$

Affine arithmetic: handling of roundoff errors

Comba, de Figueiredo, Stolfi

$$\alpha + \beta\hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error. Ex.: $\beta \times x_i$.

Roundoff error computed as:
$e = \max(\mathrm{RU}(\beta \times x_i) - \mathrm{RN}(\beta \times x_i), \mathrm{RN}(\beta \times x_i) - \mathrm{RD}(\beta \times x_i))$.

Roundoff errors accumulated in $\epsilon_\pm$:

$$\epsilon_\pm = \mathrm{RU}(\epsilon_\pm + e).$$

Affine arithmetic: handling of roundoff errors

Comba, de Figueiredo, Stolfi

$$\alpha + \beta\hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error. Ex.: $\beta \times x_i$.

Roundoff error computed as:
$e = \max(\mathrm{RU}(\beta \times x_i) - \mathrm{RN}(\beta \times x_i), \mathrm{RN}(\beta \times x_i) - \mathrm{RD}(\beta \times x_i))$.

Roundoff errors accumulated in $\epsilon_\pm$:

$$\epsilon_\pm = \mathrm{RU}(\epsilon_\pm + e).$$

Affine arithmetic: handling of roundoff errors

Hansen, Messine

$$\alpha + \beta\hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error. Ex.: $\alpha + y_0$.

Actual value belongs to the interval $[\mathrm{RD}(\alpha + y_0), \mathrm{RU}(\alpha + y_0)]$.

Coefficients are intervals:

$$\hat{\mathbf{x}} = \mathbf{x_0} + \sum_i \mathbf{x_i}\epsilon_i \text{ where } \mathbf{x_i} \in \mathbb{IR} \text{ and } \epsilon_i \text{ lives in } [-1, 1],$$

with operations performed using interval arithmetic.

Affine arithmetic: handling of roundoff errors

Rump in IntLab v.8, Kashiwagi

$$\alpha + \beta \hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error. Ex.: $\beta \times x_i$.

Roundoff errors accumulated in $\epsilon_\pm$:

$$\epsilon_\pm = \mathrm{RU}(\epsilon_\pm + e).$$

Roundoff error computed as ???

Agenda

Bounding roundoff errors

$$\alpha + \beta\hat{\mathbf{x}} + \hat{\mathbf{y}} = (\alpha + \beta x_0 + y_0) + \sum_i (\beta x_i + y_i)\epsilon_i.$$

For each operation, either $+$ or $\times$: roundoff error.

Each roundoff error is bounded: approach à la COSY (u is the machine roundoff unit):

- error on $a + b$ is less than $2u \max(|a|, |b|)$;
- error on $a \times b$ is less than $2u\mathrm{RN}(|a \times b|)$.

Roundoff errors accumulated in I:

I is an interval coefficient corresponding to $\epsilon_\pm$.

$$\mathbf{I} = \mathbf{I} + \text{ bounds on roundoff errors}$$

with operations performed using interval arithmetic.

Computing the roundoff errors

Two useful properties of floating-point arithmetic:

- in rounding-to-nearest, the roundoff error for $+$, $-$, $\times$ is a floating-point number;
- this error can be computed using floating-point arithmetic. Codes that transform $a \diamond b$ into $r + e$ with $r = \mathrm{RN}(a \diamond b)$ and e the roundoff error are called **EFT: Error Free Transforms**.

Let's make use of EFT!

Computing the roundoff errors

Two useful properties of floating-point arithmetic:

- in rounding-to-nearest, the roundoff error for $+$, $-$, $\times$ is a floating-point number;
- this error can be computed using floating-point arithmetic. Codes that transform $a \diamond b$ into $r + e$ with $r = \mathrm{RN}(a \diamond b)$ and e the roundoff error are called **EFT: Error Free Transforms**.

Let's make use of EFT!

Computing the roundoff errors

Two useful properties of floating-point arithmetic:

- in rounding-to-nearest, the roundoff error for $+$, $-$, $\times$ is a floating-point number;
- this error can be computed using floating-point arithmetic. Codes that transform $a \diamond b$ into $r + e$ with $r = \mathrm{RN}(a \diamond b)$ and e the roundoff error are called **EFT: Error Free Transforms**.

Let's make use of EFT!

Computing the roundoff errors

Two useful properties of floating-point arithmetic:

- in rounding-to-nearest, the roundoff error for $+$, $-$, $\times$ is a floating-point number;
- this error can be computed using floating-point arithmetic. Codes that transform $a \diamond b$ into $r + e$ with $r = \mathrm{RN}(a \diamond b)$ and e the roundoff error are called **EFT: Error Free Transforms**.

Let's make use of EFT!

EFT for $+$

TwoSum: $\mathrm{s} + \mathrm{e} = a + b$

$\mathrm{s} = \mathtt{RN}(a + b)$
$a' = \mathtt{RN}(\mathrm{s} - b)$
$b' = \mathtt{RN}(\mathrm{s} - a')$
$\delta_a = \mathtt{RN}(a - a')$
$\delta_b = \mathtt{RN}(b - b')$
$\mathrm{e} = \mathtt{RN}(\delta_a + \delta_b)$

The equality $s + e = a + b$ holds in **exact arithmetic**.

EFT for $\times$

FMA: Fused Multiply and Add is a floating-point operator that performs $a \times b + c$ with only one roundoff error of the exact result.

TwoProd: $\mathtt{p} + \mathsf{e} = a \times b$
$\mathtt{p} = \mathtt{RN}(a \times b)$
$\mathsf{e} = \mathtt{RN}(\mathrm{FMA}(a, b, -p))$

Without FMA: it is also possible to compute e, the code is a bit longer (17 operations).

EFT for other operations?

What about other operations? $/, \sqrt{\ }$

No EFT.

The "remainder" is a floating-point number that can be computed. . . but not very useful for our purpose.

Roundoff errors are bounded.

Accumulating the roundoff errors

Roundoff errors are computed exactly. What to do with them?

Accumulate.

How?

- No need to be very accurate.
- Need to get an upper bound.

As only non-negative values are accumulated, this gives a bound:

$$\mathrm{accu} = \mathrm{RU}(\mathrm{accu} + e).$$

Directed rounding modes or not?

Directed rounding modes can incur time penalty.

From 10 to 100 when rounding modes are set using global registers and pipelines must be flushed when the rounding mode changes.

Nothing when rounding modes are set in the instruction code (cf. CUDA for GPU, assembler for Itanium).

Directed rounding modes cannot be set. In OpenMP, OpenCL, BLAS. . . : the only rounding mode is rounding-to-nearest.

Get free of the rounding modes!

As only non-negative terms are accumulated, this gives a bound:

$$\mathrm{accu} = \mathrm{RN}((1 + 4u) \times (\mathrm{accu} + e)).$$

Directed rounding modes or not?

Directed rounding modes can incur time penalty.
From 10 to 100 when rounding modes are set using global registers and pipelines must be flushed when the rounding mode changes.
Nothing when rounding modes are set in the instruction code (cf. CUDA for GPU, assembler for Itanium).

Directed rounding modes cannot be set. In OpenMP, OpenCL, BLAS. . . : the only rounding mode is rounding-to-nearest.

Get free of the rounding modes!
As only non-negative terms are accumulated, this gives a bound:

$$\mathrm{accu} = \mathrm{RN}((1 + 4u) \times (\mathrm{accu} + e)).$$

Directed rounding modes or not?

Directed rounding modes can incur time penalty.
From 10 to 100 when rounding modes are set using global registers and pipelines must be flushed when the rounding mode changes.
Nothing when rounding modes are set in the instruction code (cf. CUDA for GPU, assembler for Itanium).

Directed rounding modes cannot be set. In OpenMP, OpenCL, BLAS. . . : the only rounding mode is rounding-to-nearest.

Get free of the rounding modes!
As only non-negative terms are accumulated, this gives a bound:

$$\mathrm{accu} = \mathrm{RN}((1 + 4u) \times (\mathrm{accu} + e)).$$

Directed rounding modes or not?

Directed rounding modes can incur time penalty.

From 10 to 100 when rounding modes are set using global registers and pipelines must be flushed when the rounding mode changes.
Nothing when rounding modes are set in the instruction code (cf. CUDA for GPU, assembler for Itanium).

Directed rounding modes cannot be set. In OpenMP, OpenCL, BLAS. . . : the only rounding mode is rounding-to-nearest.

Get free of the rounding modes!

As only non-negative terms are accumulated, this gives a bound:

$$\mathrm{accu} = \mathrm{RN}((1 + 4u) \times (\mathrm{accu} + e)).$$

Agenda

Framework

Variants of affine arithmetic are implemented in **IBEX**:

- double = no guarantee, arithmetic provided by the processor;
- AF_no: affine arithmetic without control of roundoff errors;
- sAF: de Figueiredo and Stolfi's version;
- iAF: Hansen's and Messine's variant: affine arithmetic with interval coefficients;
- fAF: roundoff errors are bounded and accumulated in an interval;
- fAF_v2: roundoff errors are computed and accumulated – in rounding to nearest – in a floating-point value.

Framework

Variants of affine arithmetic are implemented in **IBEX**:

- double = no guarantee, arithmetic provided by the processor;
- AF_no: affine arithmetic without control of roundoff errors;
- sAF: de Figueiredo and Stolfi's version;
- iAF: Hansen's and Messine's variant: affine arithmetic with interval coefficients;
- fAF: roundoff errors are bounded and accumulated in an interval;
- fAF_v2: roundoff errors are computed and accumulated – in rounding to nearest – in a floating-point value.

Evaluation of Shekel 5: time and accuracy

Points	x_1	$\mathbf{x}_2$	$f(\mathbf{x}_2)$
double	0.12s	-	-
Interval	148.4s	154.0s	$[-10.1663975282993153, -7.28436947861868234]$
AF_no	1072.5s	1377.1s	$[-10.161132318290755\mathbf{8}, -7.0491294418416270\mathbf{4}]$
sAF	3096.5s	3593.6s	$[-10.161132318290800\mathbf{2}, -7.04912944184159329]$
iAF	1898.4s	3763.7s	$[-10.161132318290762\mathbf{9}, -7.04912944184162082]$
fAF	1900.1s	1946.8s	$[-10.161132318291230\mathbf{1}, -7.04912944184125667]$
fAF_v2	1306.3s	1579.4s	$[-10.161132318290791\mathbf{3}, -7.04912944184159862]$

Table: **CPU-time of 10^8 evaluations of the Shekel-5 function.**

Caution

Still experimental code:

- compiler's options probably need a finer tuning;
- with FMA: prototype machine, buggy code for the time being.

Evaluation of Shekel 5: timings

Points	x_1	x_2
double	0.01s	-
Interval	1.48s	1.54s
Interval	8.45s	9.86s
AF_no	10.72s	13.77s
AF_no	42.5s	57.6s
sAF	30.96s	35.94s
sAF	562.9s	371.6s
iAF	18.98s	37.64s
iAF	73.6s	141.7s
fAF	19.00s	19.47s
fAF	75.0s	97.2s
fAF_v2	13.06s	15.79s
fAF_v2	71.8s	95.6s

$\simeq$ **CPU-time of 10^6 evaluations of the Shekel-5 function on a Xeon**
CPU-time of 10^6 evaluations of the Shekel-5 function on an AMD (with FMA)

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF (bounds on the roundoff errors, only one interval: the remainder):

- on Xeon: 1 to 2 times faster than other variants of affine arithmetic,
  50% to 100% slower than affine arithmetic without roundoff errors,
  10,000 times slower than double arithmetic;
- on AMD: up to 7 times faster than other variants of affine arithmetic,
  twice slower than affine arithmetic without roundoff errors,
  100 times slower than double arithmetic;
- in both cases, 10 times slower than interval arithmetic.

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF (bounds on the roundoff errors, only one interval: the remainder):

- on Xeon: 1 to 2 times faster than other variants of affine arithmetic,
  50% to 100% slower than affine arithmetic without roundoff errors,
  10,000 times slower than double arithmetic;
- on AMD: up to 7 times faster than other variants of affine arithmetic,
  twice slower than affine arithmetic without roundoff errors,
  100 times slower than double arithmetic;
- in both cases, 10 times slower than interval arithmetic.

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF_v2 (computations of the roundoff errors, no changes of the rounding modes):

- on Xeon: from 50% to 2.5 times faster than other variants of affine arithmetic,
  slightly slower than affine arithmetic without roundoff errors,
  1,500 times slower than double arithmetic;
- on AMD: from 2 to 10 times faster than other variants of affine arithmetic,
  similar to affine arithmetic without roundoff errors, 50 times slower than double arithmetic;
- in both cases, 6 times slower than interval arithmetic.

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF_v2 (computations of the roundoff errors, no changes of the rounding modes):

- on Xeon: from 50% to 2.5 times faster than other variants of affine arithmetic,
  slightly slower than affine arithmetic without roundoff errors,
  1,500 times slower than double arithmetic;
- on AMD: from 2 to 10 times faster than other variants of affine arithmetic,
  similar to affine arithmetic without roundoff errors, 50 times slower than double arithmetic;
- in both cases, 6 times slower than interval arithmetic.

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF_v2 (computations of the roundoff errors, no changes of the rounding modes):

- on Xeon: from 50% to 2.5 times faster than other variants of affine arithmetic,
  slightly slower than affine arithmetic without roundoff errors,
  1,500 times slower than double arithmetic;
- on AMD: from 2 to 10 times faster than other variants of affine arithmetic,
  similar to affine arithmetic without roundoff errors, 50 times slower than double arithmetic;
- in both cases, 6 times slower than interval arithmetic.

On different architectures

Caution (reminder): experimental and probably buggy code.

Use of fAF_v2 (computations of the roundoff errors, no changes of the rounding modes):

- on Xeon: from 50% to 2.5 times faster than other variants of affine arithmetic,
  slightly slower than affine arithmetic without roundoff errors,
  1,500 times slower than double arithmetic;
- on AMD: from 2 to 10 times faster than other variants of affine arithmetic,
  similar to affine arithmetic without roundoff errors, 50 times slower than double arithmetic;
- in both cases, 6 times slower than interval arithmetic.

Agenda

Conclusion and future work

For an implementation of interval/affine/... arithmetic using floating-point arithmetic:

- to preserve the inclusion property: handle roundoff errors;
- many ways to handle roundoff errors: bound them, compute them:
  think in terms of significant bits
- influence of the architecture and of the instruction set: to be further explored.

Conclusion and future work

For an implementation of interval/affine/...arithmetic using floating-point arithmetic:

- to preserve the inclusion property: handle roundoff errors;
- many ways to handle roundoff errors: bound them, compute them:
  think in terms of significant bits
- influence of the architecture and of the instruction set: to be further explored.

Conclusion and future work

For an implementation of interval/affine/. . . arithmetic using floating-point arithmetic:

- to preserve the inclusion property: handle roundoff errors;
- many ways to handle roundoff errors: bound them, compute them:
  think in terms of significant bits
- influence of the architecture and of the instruction set: to be further explored.

Global optimization

IBEX has code to solve global optimization problems: technique chosen here = Affine Relaxation Technique.

COCONUT has benchmark problems: 113 problems are tested here.

Global optimization

ersion of Affine rithmetic used in *IBEX*	non-reliable		reliable							
	AF_no		sAF		iAF		fAF		fAF_v2	
	nb	t (s)	nb	t (s)	nb	t (s)	nb	t (s)	nb	t (s)
Average for olved problems	111	72	113	73	113	72	113	67	113	65
erage for problems ved by all versions	111	72	111	75	111	73	111	69	111	66