Verified Numerical Computation and kv Library

Masahide Kashiwagi
kashi@waseda.jp
http://verifiedby.me/

Waseda University, Japan

(Apr. 10, 2019)

Outline

- error in numerical computation
- verified numerical computation
- interval arithmetic
- Krawczyk method
- introduction of kv library
    - outline
    - interval arithmetic
    - automatic differentiation, double-double arithmeric, mpfr, affine arithmeric
    - power series arithmetic (psa)
    - quadrature
    - ordinary differential equations

Numerical Computation

Equations can not be solved

Human beings have discovered a methodology to describe phenomena by equations, which made it possible to create various technologies and predict the future. However, almost all of equations can not be solved in the form of simple expressions.

Numerical Computation

To solve "mathematical problems that can not be solved analytically" by computer.

- Numerical calculations are used in various fields along with the development of computers, which is indispensable to modern society.
  - weather forecast, strength calculation of buildings, car crashes, chemical reaction, fluid simulation
  - calculation of trajectories of celestial bodies, rockets, etc.
  - computer graphics
- The world's first computer, ENIAC, was depeloped with the objective of numerical calculation of missile trajectory.

Representation of real numbers in computer (IEEE 754 double precision floating point number)

representation of real numbers

- store real numbers in the form of "(number close to 1) $\times$ (power number)" such as 6.02×10^{23}.
- stored in binary numbers and use 64bits=8bytes.
- 64bits are divided as follows:

s	e_{10}	e_9	$\cdots$	e_0	m_{51}	m_{50}	$\cdots$	m_0

e : binary integer ($e = \sum_{i=0}^{10} e_i 2^i$),

m : binary fraction ($m = \sum_{i=0}^{51} m_i 2^{i-52}$)

corresponds to the value represented by the following:

$$(-1)^s \times (1 + m) \times 2^{e-1023}$$

- accuracy(significant digit) is about 16 digits.
$2^{-52} \simeq 2.22 \times 10^{-16} \simeq 10^{-15.65}$

Detail of IEEE 754 double precision format

Detail of IEEE 754 double precision format

s	e_{10}	e_9	$\cdots$	e_0	m_{51}	m_{50}	$\cdots$	m_0

$$e = \sum_{i=0}^{10} e_i 2^i, \quad m = \sum_{i=0}^{51} m_i 2^{i-52}$$

	$m = 0$	$m \neq 0$
$e = 0$	± 0	$(-1)^s \times (0 + m) \times 2^{-1022}$ (denormalized number)
$1 \leq e \leq 2046$	$(-1)^s \times (1 + m) \times 2^{e-1023}$ (normalized number)	
$e = 2047$	$\pm\infty$	NaN (Not a Number)

0	$2^{-1074} \leq x < 2^{-1022}$	$2^{-1022} \leq x < 2^{1024}$	$+\infty$
	denormalized number	normalized number	

Rounding Error

Rounding Error

- Since the siginificant digit is about 16 digits, when we calculate $1 \div 3$, for example, we have some error as:

$$1 \div 3 = 0.333333333333333148\ldots$$

  This is called rounding error.

- Since binary numbers are used, even $1 \div 10 = 0.1$, an error occurs as:

$$1 \div 10 = 0.10000000000000000005551\ldots$$

- The error is no so large if it is calculated only once, but if errors accumulate, the error may expand more than we imagine.

rounding error of quadratic equations

quadratic equation

The solution of the quadratic equation $ax^2 + bx + c = 0$ can be written as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

example of large errors

$$a = 1,\ b = 10^{15},\ c = 10^{14}$$

- (The quadratic formula) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a} = -0.125$
- (rationalizing numerator)
  $\dfrac{2c}{-b - \sqrt{b^2 - 4ac}} = -0.10000000000000002$

error of linear equations

linear equation

$$\begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

true solution

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 205117922 \\ 83739041 \end{pmatrix}$$

solution by Gaussian elimination

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 106018308.0071325 \\ 43281793.001783125 \end{pmatrix}$$

calculating residual of wrong solution$\cdots$

$$\begin{pmatrix} 64919121 & -159018721 \\ 41869520.5 & -102558961 \end{pmatrix} \begin{pmatrix} 106018308.0071325 \\ 43281793.001783125 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} ??$$

Patriot Missile Accedent in the Gulf War

`http://www-users.math.umn.edu/~arnold/disasters/patriot.html`
`http://www.fas.org/spp/starwars/gao/im92026.htm`

In the Gulf war, the Patriot missile of USA was used for the interception of the Iraqi Scud missile. However, due to a small error of the internal computer, the Patriot missile failed to track and intercept the Scud missile.

- Date: Feb. 25, 1991
- Place: Dharan, Saudi Arabia
- Damage: 28 soldiers killed, around 100 other people injured

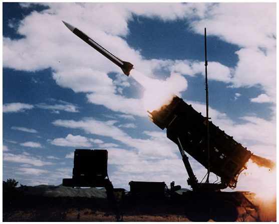


Figure 3: Correctly Calculated Range Gate

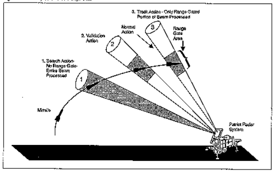
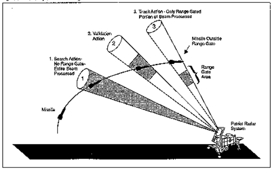

3. Track Action - Only Range Gated Portion of Beam Processed
Normal Action
Range Gate Area
2. Validation Action
1. Search Action- No Range Gate- Entire Beam Processed
Missile
Patriot Radar System

Figure 5: Incorrectly Calculated Range Gate


3. Track Action - Only Range Gated Portion of Beam Processed
2. Validation Action
Missile Outside Range Gate
1. Search Action- No Range Gate- Entire Beam Processed
Range Gate Area
Missile
Patriot Radar System

Verified Numerical Computation

Verified Numerical Computation

Numerical computation method which calculates not only approximate solution but mathematically rigorous error estimation of the solution.

Required Technology

- Interval Arithmeric (estimates the rounding error in the calculation using round-up and round-down. also evaluates the image of the function.)
- Fixed Point Theorem (By confirming the sufficient condition of the fixed point theorem by interval arithmetic, we guarantee the existence and the range of the solution of the equation.)
- Automatic Differentiation is also important.

Interval Arithmetic (1/6)

Interval Arithmeric

- Interval arithmetic is the most fundamental technique for verified numerical computation.
- In interval arithmetic, numerical values are expressed by closed interval $X = [a, b]$ where the endpoints are floating point numbers that can be represented by a computer.
- In interval arithmeric, the operation between intervals is performed so as to include all possible calculation results for the values included in the interval operand.
- In interval arithmeric, generally, directed rounding defined by IEEE 754 standard is used.
- There are two rolls of interval arithmeric, estimation of rounding error and evaluation of range of function.

Interval Arithmeric (2/6)

- $X = [a, b]$, $Y = [c, d]$
- $\underline{\cdot}$ and $\overline{\cdot}$ means "round to $-\infty$" and "round to $+\infty$", respectively.

- Addition $X + Y = [a \underline{+} c, b \overline{+} d]$
- Subtraction $X - Y = [a \underline{-} d, b \overline{-} c]$

- Multiplication $X \times Y =$

	$d < 0$	$c \le 0, d \ge 0$	$c > 0$
$b < 0$	$[b \underline{\times} d, a \overline{\times} c]$	$[a \underline{\times} d, a \overline{\times} c]$	$[a \underline{\times} d, b \overline{\times} c]$
$a \le 0, b \ge 0$	$[b \underline{\times} c, a \overline{\times} c]$	$[\min(a \underline{\times} d, b \underline{\times} c), \max(a \overline{\times} c, b \overline{\times} d)]$	$[a \underline{\times} d, b \overline{\times} d]$
$a > 0$	$[b \underline{\times} c, a \overline{\times} d]$	$[b \underline{\times} c, b \overline{\times} d]$	$[a \underline{\times} c, b \overline{\times} d]$

- Division $X/Y =$

	$d < 0$	$c > 0$
$b < 0$	$[b \underline{/} c, a \overline{/} d]$	$[a \underline{/} c, b \overline{/} d]$
$a \le 0, b \ge 0$	$[b \underline{/} d, a \overline{/} d]$	$[a \underline{/} c, b \overline{/} c]$
$a > 0$	$[b \underline{/} d, a \overline{/} c]$	$[a \underline{/} d, b \overline{/} c]$

(defined only when $Y \not\ni 0$)

- Square Root $\sqrt{X} = [\underline{\sqrt{a}}, \overline{\sqrt{b}}]$

Interval Arithmetic (3/6)

If we calculate $(1 \div 3) \times 3$ with 3-digits decimal number $\cdots$

- Numerical Calculation (Not Verified)

$$1 \div 3 = 0.333$$
$$0.333 \times 3 = 0.999$$

Error due to rounding off is obserbed.

- Interval Arithmetic
Start from an interval with no width.

$$[1, 1] \div [3, 3] = [0.333, 0.334]$$
$$[0.333, 0.334] \times [3, 3] = [0.999, 1.01]$$

Calculation proceeds so as to always include the true value in the interval.

Solving previous example by interval arithmetic

quadratic equation

The solution of the quadratic equation $ax^2 + bx + c = 0$ can be written as:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

example of large errors

$$a = 1,\ b = 10^{15},\ c = 10^{14}$$

- (The quadratic formula) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a} = [-0.1875, -0.0625]$
- (rationalizing numerator) $\dfrac{2c}{-b - \sqrt{b^2 - 4ac}} =$
  $[-0.10000000000000004, -0.099999999999999991]$

Interval Arithmeric (4/6)

How to change rounding direction

- IEEE 754 Standard requests that the rounding direction can be changed. However, the method of changing rounding direction differs on CPU and compiler.
- In the case of Intel X86 cpu, the method of changing rounding direction is complicated, for example, the method is different between FPU and SSE2.
- Recently, since C99 compliant compiler became popular, using `fenv.h` and `fesetround` absorbs the hardware difference of changing rounding direction.

```
#include <iostream>
#include <fenv.h>
int main()
{
    double x=1, y=10, z;
    std::cout.precision(17);

    fesetround(FE_TONEAREST);
    z = x / y;
    std::cout << z << std::endl;
    fesetround(FE_DOWNWARD);
    z = x / y;
    std::cout << z << std::endl;
    fesetround(FE_UPWARD);
    z = x / y;
    std::cout << z << std::endl;
}
```

```
0.10000000000000001
0.099999999999999992
0.10000000000000001
```

Interval Arithmeric (5/6)

Things to be careful on implementation of interval arithmetic

- (Input Error) We can not control rounding direction of constants in program text (for example, "`0.1`" of `double x = 0.1;`) because the conversion from the decimal string to the binary number is executed at compile time.
- (Output Error) Similarly, we can not control rounding direction in the display of numbers such as "`std::cout << x << std::endl;`".
- (Compiler Optimization) Compiler optimizations such as "-O3" may change the order of calculations, and the rounding direction may not be changed as intended. $\Longrightarrow$ necessary to suppress optimization appropriately by using `volatile` etc.
- (Mathematical Functions) Only `sqrt` has guaranteed accuracy and can change the rounding direction, and mathematical functions other than `sqrt` can not be trusted at all.

Interval Arithmeric (6/6)

Overestimation of interval arithmetic

Although the interval operation certainly includes true values, the interval width may become wider than expected.

$$f(x) = x^2 - 2x, \quad x \in [0.9, 1.1]$$

The result of the interval arithmetic is $[-1.39, -0.59]$, but the true image is $[-1, -0.99]$.

Supressing overestimation

Mean Value Form Rather than evaluationg $f(I)$ directly, is is often better to calculate

$$f(c) + f'(I)(I - c)$$

with $c = \mathrm{mid}(I)$.

Affine Arithmetic (later mentioned)

Krawczyk method (Krawczyk(1969), Kahan(1968))

A Method to guarantee the existence of a solution of nonlinear equation $f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n$.

Krawczyk method

Let $I \subset \mathbb{R}^n$ be interval vector (candidate set), $c = \mathrm{mid}(I)$, $R \simeq f'(c)^{-1}$, E be identity matrix, and

$$K(I) = c - Rf(c) + (E - Rf'(I))(I - c) \quad .$$

If $K(I) \subset \mathrm{int}(I)$ then there exists a unique solution of $f(x) = 0$ in I . ($\mathrm{int}(I)$: interior of I)

Proof

Apply mean value form and contraction mapping theorem to $g(x) = x - Rf(x)$.

Krawczyk method has $f'(I)$ (interval matrix which encloses all Jacobi matrix of f in I) $\Longrightarrow$ Automatic differentiation is required

Example of Krawczyk method

Example

Show existence of a solution of equation $\begin{cases} x_1^2 + x_2^2 - 1 = 0 \\ x_1 - x_2 = 0 \end{cases}$ in interval vector $I = \begin{pmatrix} [0.6, 0.8] \\ [0.6, 0.8] \end{pmatrix}$.

Calculation Example

Let $c = \begin{pmatrix} 0.7 \\ 0.7 \end{pmatrix}$, $R = \begin{pmatrix} 0.4 & 0.5 \\ 0.4 & -0.5 \end{pmatrix} \simeq f'(c)^{-1}$. Then $f'(I) = \begin{pmatrix} [1.2, 1.6] & [1.2, 1.6] \\ 1 & -1 \end{pmatrix}$ and

$$c - Rf(c) + (E - Rf'(I))(I - c) = \begin{pmatrix} [0.68, 0.736] \\ [0.68, 0.736] \end{pmatrix} \subset \begin{pmatrix} [0.6, 0.8] \\ [0.6, 0.8] \end{pmatrix}$$

. This guarantees existence of unique solution.

The advantage of Krawczyk method is that it is composed by only automatic procedures.

Krawczyk 法の応用

近似解 c を元にした解の存在保証

$R \simeq f'(c)^{-1}$、$r = 2\|Rf(c)\|$ (Newton 法の修正量の 2 倍) とし、

$$I = c + r \begin{pmatrix} [-1,1] \\ \vdots \\ [-1,1] \end{pmatrix}$$

を候補者集合として Krawczyk 法を使うとよい。

区間ベクトル I 内の全解探索

- 区間ベクトル I での解の存在定理 (Krawczyk 法)
- 区間ベクトル I での解の非存在定理 (例えば $f(I) \not\ni 0$ なら I に解なし)

の 2 つの定理を両方試し、両方共失敗したならば区間を分割する、という作業をどちらかが成立するまで再帰的に繰り返す。

kv – a C++ library for verified numerical computation

Overview of kv library

- Available to download at http://verifiedby.me/kv/.
- Development of the library began in autumn 2007. The initial release of the library was Sep. 18, 2013. The latest release is version 0.4.48.
- Written in C++. The boost C++ Library is required.
- Header-only. kv library is designed to work without "install" but only with the header files in itself.
- Open source. If we assert that the result of verified numerical computation is 'proof', all programs used for computation must be public.
- Data type of numbers in the calculation is not restrected to "double". Data type can be easily changed using "template" feature in C++.

Numeric Data types available in kv library

Numeric data types available in kv library

- double
- interval (with many verified mathematical functions)
- double-double
- MPFR wrapper
- complex
- automatic differentiation
- affine arithmeric
- Power Series Arithmeric (PSA)
- and these combinations

combination of numeric data types

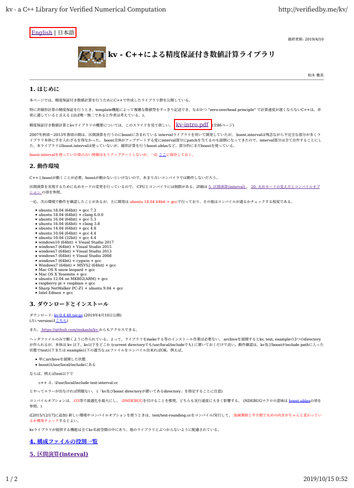
For example, the type ‘autodif<interval<dd>>’ means ‘autodif using interval using double-double’.

Applications in kv library

applications in kv library

- verified solution of nonlinear equations by Krawczyk method
- finding all solutions of nonlinear equations
- initial value problems of ordinary differential equations
- boundary value problems of ordinary differential equations
- numerical integration for 1d and 2d functions
- numerical integration for functions with endpoint singularity
- special functions such as gamma, bessel, etc.
- verification of optimization problem by KKT equation
- etc.

kv web page (in Japanese)



kv - a C++ Library for Verified Numerical Computation http://verifiedby.me/kv/

English | 日本語

最終更新: 2019/4/10

kv - C++による精度保証付き数値計算ライブラリ

柏木 雅英

1. はじめに

本ページでは、精度保証付き数値計算を行うためにC++で作成したライブラリ群を公開している。

特に非線形計算の精度保証を行うとき、template機能によって複雑な数値型をすっきり記述でき、なおかつ "zero-overhead principle" で計算速度が遅くならないC++は、非常に適していると言える (ほぼ唯一無二であると作者は考えている。)。

精度保証付き数値計算とkvライブラリの概要については、このスライドを見て欲しい。 kv-intro.pdf (全86ページ)

2007年秋頃～2013年春頃の間は、区間演算を行うのにboostに含まれている intervalライブラリを用いて開発していたが、 boost.intervalは残念ながら不完全な部分が多くライブラリ本体に手を入れざるを得なかった。 boost全体がアップデートする度にinterval部分にpatchを当てるのも面倒になってきたので、interval部分は全て自作することにした。本ライブラリはboost.intervalは使っていないが、線形計算を行うboost.ublasなど、部分的にまだboostを使っている。

boost.intervalを使っていた頃の古い情報はもうアップデートしないが、一応 ここに保存しておく。

2. 動作環境

C++とboostが動くことが必要。boostが動かないといけないので、あまり古いコンパイラでは動作しないだろう。

区間演算を実現するために丸めモードの変更を行っているので、 CPUとコンパイラには制限がある。詳細は 5. 区間演算(interval) 、 20. 丸めモードの変え方とコンパイルオプション の項を参照。

一応、次の環境で動作を確認したことがあるが、主に開発は ubuntu 18.04 64bit + gccで行っており、その他はコンパイルが通るかチェックする程度である。

- ubuntu 18.04 (64bit) + gcc 7.3
- ubuntu 18.04 (64bit) + clang 6.0.0
- ubuntu 16.04 (64bit) + gcc 5.3
- ubuntu 16.04 (64bit) + clang 3.8
- ubuntu 14.04 (64bit) + gcc 4.8
- ubuntu 10.04 (64bit) + gcc 4.4
- ubuntu 10.04 (32bit) + gcc 4.4
- windows10 (64bit) + Visual Studio 2017
- windows7 (64bit) + Visual Studio 2015
- windows7 (64bit) + Visual Studio 2013
- windows7 (64bit) + Visual Studio 2008
- windows7 (64bit) + cygwin + gcc
- Windows7 (64bit) + MSYS2 (64bit) + gcc
- Mac OS X snow leopard + gcc
- Mac OS X Yosemite + gcc
- ubuntu 12.04 on MK802(ARM) + gcc
- raspberry pi + raspbian + gcc
- Sharp NetWalker PC-Z1 + ubuntu 9.04 + gcc
- Intel Edison + gcc

3. ダウンロードとインストール

ダウンロード: kv-0.4.48.tar.gz (2019年4月10日公開)
(古いversionはこちら)

また、 https://github.com/mskashi/kv からもアクセスできる。

ヘッダファイルのみで動くように作られている。よって、ライブラリをmakeする等のインストール作業は必要ない。 archiveを展開するとkv, test, exampleの3つのdirectoryが作られるが、本体は kv 以下。kv以下をどこか (current directoryでも/usr/local/includeでも) に置いておくだけで良い。動作確認は、kv及びboostがinclude pathに入った状態でtest以下または example以下の適当な.ccファイルをコンパイル出来ればOK。例えば、

- 単にarchiveを展開した状態
- boostは/usr/local/includeにある

ならば、例えばtest以下で

```
c++ -I.. -I/usr/local/include test-interval.cc
```

とやってエラーが出なければ問題ない。 (「kv及びboost directoryが置いてあるdirectory」を指定することに注意)

コンパイルオプションは、-O3等で最適化を最大にし、 -DNDEBUGを付けることを推奨。どちらも実行速度に大きく影響する。 (NDEBUGマクロの意味は boost.ublasの項を参照。)

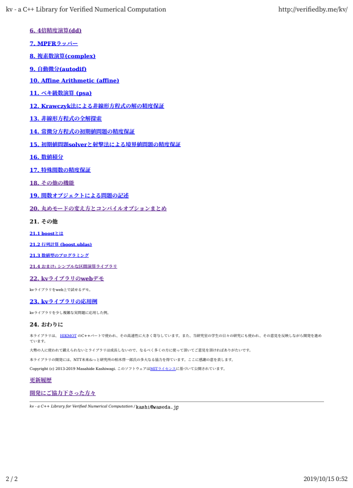
([2015/12/17]に追加) 新しい環境やコンパイルオプションを使うときは、test/test-rounding.ccをコンパイル/実行して、 加減乗除と平方根で丸めの向きがちゃんと変わっているか簡易チェックするとよい。

kvライブラリが提供する機能は全てkv名前空間の中にあり、他のライブラリとぶつからないように配慮されている。

4. 構成ファイルの役割一覧

5. 区間演算(interval)

1 / 2 2019/10/15 0:52



kv - a C++ Library for Verified Numerical Computation http://verifiedby.me/kv/

6. 4倍精度演算(dd)

7. MPFRラッパー

8. 複素数演算(complex)

9. 自動微分(autodif)

10. Affine Arithmetic (affine)

11. ベキ級数演算 (psa)

12. Krawczyk法による非線形方程式の解の精度保証

13. 非線形方程式の全解探索

14. 常微分方程式の初期値問題の精度保証

15. 初期値問題solverと射撃法による境界値問題の精度保証

16. 数値積分

17. 特殊関数の精度保証

18. その他の機能

19. 関数オブジェクトによる問題の記述

20. 丸めモードの変え方とコンパイルオプションまとめ

21. その他

21.1 boostとは

21.2 行列計算 (boost.ublas)

21.3 数値型のプログラミング

21.4 おまけ: シンプルな区間演算ライブラリ

22. kvライブラリのwebデモ

kvライブラリをweb上で試せるデモ。

23. kvライブラリの応用例

kvライブラリを少し複雑な実問題に応用した例。

24. おわりに

本ライブラリは、 HIKMOT のC++パートで使われ、その高速性に大きく寄与しています。また、当研究室の学生の日々の研究にも使われ、その意見を反映しながら開発を進めています。

大勢の人に使われて鍛えられないとライブラリは成長しないので、なるべく多くの方に使って頂いてご意見を頂ければありがたいです。

本ライブラリの開発には、NTT未来ねっと研究所の柏木啓一郎氏の多大なる協力を得ています。ここに感謝の意を表します。

Copyright (c) 2013-2019 Masahide Kashiwagi. このソフトウェアはMITライセンスに基づいて公開されています。

更新履歴

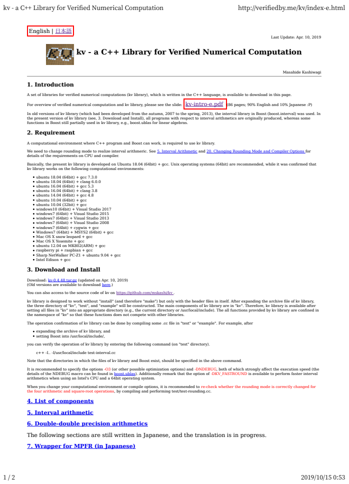
開発にご協力下さった方々

kv - a C++ Library for Verified Numerical Computation / kashi@waseda.jp

2 / 2 2019/10/15 0:52

http://verifiedby.me/kv/

kv web page (in English)



kv - a C++ Library for Verified Numerical Computation   http://verifiedby.me/kv/index-e.html

English | 日本語

Last Update: Apr. 10, 2019

kv - a C++ Library for Verified Numerical Computation

Masahide Kashiwagi

1. Introduction

A set of libraries for verified numerical computations (kv library), which is written in the C++ language, is available to download in this page.

For overview of verified numerical computation and kv library, please see the slide: kv-intro-e.pdf (86 pages; 90% English and 10% Japanese :P)

In old versions of kv library (which had been developed from the autumn, 2007 to the spring, 2013), the interval library in Boost (boost.interval) was used. In the present version of kv library (see, 3. Download and Install), all programs with respect to interval arithmetics are originally produced, whereas some functions in Boost still partially used in kv library, e.g., boost.ublas for linear algebras.

2. Requirement

A computational environment where C++ program and Boost can work, is required to use kv library.

We need to change rounding mode to realize interval arithmeric. See 5. Interval Arithmetic and 20. Changing Rounding Mode and Compiler Options for details of the requirements on CPU and compiler.

Basically, the present kv library is developed on Ubuntu 18.04 (64bit) + gcc. Unix operating systems (64bit) are recommended, while it was confirmed that kv library works on the following computational environments:

- ubuntu 18.04 (64bit) + gcc 7.3.0
- ubuntu 18.04 (64bit) + clang 6.0.0
- ubuntu 16.04 (64bit) + gcc 5.3
- ubuntu 16.04 (64bit) + clang 3.8
- ubuntu 14.04 (64bit) + gcc 4.8
- ubuntu 10.04 (64bit) + gcc
- ubuntu 10.04 (32bit) + gcc
- windows10 (64bit) + Visual Studio 2017
- windows7 (64bit) + Visual Studio 2015
- windows7 (64bit) + Visual Studio 2013
- windows7 (64bit) + Visual Studio 2008
- windows7 (64bit) + cygwin + gcc
- Windows7 (64bit) + MSYS2 (64bit) + gcc
- Mac OS X snow leopard + gcc
- Mac OS X Yosemite + gcc
- ubuntu 12.04 on MK802(ARM) + gcc
- raspberry pi + raspbian + gcc
- Sharp NetWalker PC-Z1 + ubuntu 9.04 + gcc
- Intel Edison + gcc

3. Download and Install

Download: kv-0.4.48.tar.gz (updated on Apr. 10, 2019)
(Old versions are available to download here.)

You can also access to the source code of kv on https://github.com/mskashi/kv .

kv library is designed to work without "install" (and therefore "make") but only with the header files in itself. After expanding the archive file of kv library, the three directory of "kv", "test", and "example" will be constructed. The main components of kv library are in "kv". Therefore, kv library is available after setting all files in "kv" into an appropriate directory (e.g., the current directory or /usr/local/include). The all functions provided by kv library are confined in the namespace of "kv" so that these functions does not compete with other libraries.

The operation confirmation of kv library can be done by compiling some .cc file in "test" or "example". For example, after

- expanding the archive of kv library, and
- setting Boost into /usr/local/include/,

you can verify the operation of kv library by entering the following command (on "test" directory).

```
c++ -I.. -I/usr/local/include test-interval.cc
```

Note that the directories in which the files of kv library and Boost exist, should be specified in the above command.

It is recommended to specify the options -O3 (or other possible optimization options) and -DNDEBUG, both of which strongly affect the execution speed (the details of the NDEBUG macro can be found in boost.ublas). Additionally remark that the option of -DKV_FASTROUND is available to perform faster interval arithmetics when using an Intel's CPU and a 64bit operating system.

When you change your computational environment or compile options, it is recommended to re-check whether the rounding mode is correctly changed for the four arithmetic and square-root operations, by compiling and performing test/test-rounding.cc.

4. List of components

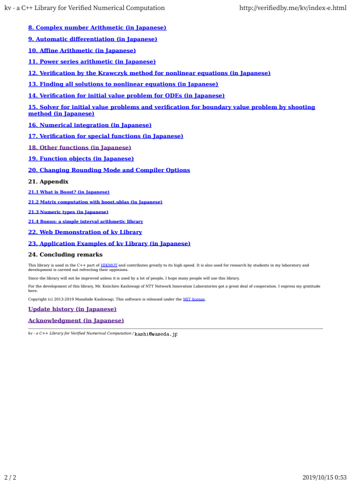
5. Interval arithmetic

6. Double-double precision arithmetics

The following sections are still written in Japanese, and the translation is in progress.

7. Wrapper for MPFR (in Japanese)

1 / 2   2019/10/15 0:53



kv - a C++ Library for Verified Numerical Computation   http://verifiedby.me/kv/index-e.html

8. Complex number Arithmetic (in Japanese)

9. Automatic differentiation (in Japanese)

10. Affine Arithmetic (in Japanese)

11. Power series arithmetic (in Japanese)

12. Verification by the Krawczyk method for nonlinear equations (in Japanese)

13. Finding all solutions to nonlinear equations (in Japanese)

14. Verification for initial value problem for ODEs (in Japanese)

15. Solver for initial value problems and verification for boundary value problem by shooting method (in Japanese)

16. Numerical integration (in Japanese)

17. Verification for special functions (in Japanese)

18. Other functions (in Japanese)

19. Function objects (in Japanese)

20. Changing Rounding Mode and Compiler Options

21. Appendix

21.1 What is Boost? (in Japanese)

21.2 Matrix computation with boost.ublas (in Japanese)

21.3 Numeric types (in Japanese)

21.4 Bonus: a simple interval arithmetic library

22. Web Demonstration of kv Library

23. Application Examples of kv Library (in Japanese)

24. Concluding remarks

This library is used in the C++ part of HIKMOT and contributes greatly to its high speed. It is also used for research by students in my laboratory and development is carried out refrecting their oppinions.

Since the library will not be improved unless it is used by a lot of people, I hope many people will use this library.

For the development of this library, Mr. Keiichiro Kashiwagi of NTT Network Innovation Laboratories got a great deal of cooperation. I express my gratitude here.

Copyright (c) 2013-2019 Masahide Kashiwagi. This software is released under the MIT license.

Update history (in Japanese)

Acknowledgment (in Japanese)

kv - a C++ Library for Verified Numerical Computation / kashi@waseda.jp

2 / 2   2019/10/15 0:53

http://verifiedby.me/kv/index-e.html

Why C++?

- For a program of the same notation (below is an exmaple),

```
y = (x+1) * (x-2) + log(x);
```

  we would like to use different numeric types that perform various special operations such as:

  - double
  - interval
  - autodif
  - autodif<interval>
  - PSA
  - MPFR
  - interval<MPFR>
  - etc.

  using operator-overloading.
- Using 'dynamically typed languages' such as python, ruby, matlab will cause speed down because type determination is done at run time.
- Using C++'s template feature, we can describe a program with 'generic type' (without assuming data type), and do not speed down because all type determination is completed at compile time.

C++ 's template function

without template

```
#include <iostream>

void swap(int& a, int& b) {
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
}

void swap(double& a, double& b) {
    double tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a=1, b=2;

    swap(a, b); // swap int value
    std::cout << a << " " << b << "\n";

    double x=1., y=2.;

    swap(x, y); // swap double value
    std::cout << x << " " << y << "\n";
}
```

with template

```
#include <iostream>

template <class T> void swap(T& a, T& b) {
    T tmp;

    tmp = a;
    a = b;
    b = tmp;
}

int main()
{
    int a=1, b=2;

    swap(a, b); // swap int value
    std::cout << a << " " << b << "\n";

    double x=1., y=2.;

    swap(x, y); // swap double value
    std::cout << x << " " << y << "\n";
}
```

C++ 's template class

without template

```
#include <iostream>

class pair_int {
    int a, b;
    public:

    pair_int(int x, int y) : a(x), b(y) {}

    void print() {
        std::cout << a << " " << b << "\n";
    }
};

class pair_double {
    double a, b;
    public:

    pair_double(double x, double y) : a(x), b
        (y) {}

    void print() {
        std::cout << a << " " << b << "\n";
    }
};

int main()
{
    pair_int p(1, 2);
    p.print();

    pair_double q(1., 2.);
    q.print();
}
```

with template

```
#include <iostream>

template <class T> class pair {
    T a, b;
    public:

    pair(T x, T y) : a(x), b(y) {}

    void print() {
        std::cout << a << " " << b << "\n";
    }
};

int main()
{
    pair<int> p(1, 2);
    p.print();

    pair<double> q(1., 2.);
    q.print();
}
```

Matrix and Vector calculation

boost.ublas

- Matrix and vector calculation in kv library depends on the 'ublas' contained in the boost library (`http://www.boost.org/`).
- Since ublas is a template library, it is possible to handle interval matrix/vector etc. naturally.
- Although the name is 'ublas', it means that it has all functions of BLAS, it is not fast like BLAS.

kv ライブラリにおける線形計算

- 線形計算においては、例えば行列積 $C = A \times B$ を
  1. 丸めの向きを下向きに変更してから $\underline{C} = A \times B$ を計算
  2. 丸めの向きを上向きに変更してから $\overline{C} = A \times B$ を計算

  のような手順で計算することによって、丸めの向きの変更回数を減らし高速な BLAS を利用できる。
- KV ライブラリでは double 以外の型を自然に利用できることを重視したため、現在の version ではこのような技術は全く使われていない

Interval Arithmetic (interval)

- provide inf-sup type interval arithmetic.
- provide verified mathematical functions such as exp, log, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, asinh, acosh, atanh, expm1, log1p, abs, pow .
- provide mutual conversion function between `double` and decimal strings with directed rounding.
- The types of lower/upper bounds are templates, and we can use numeric types other than double such as double-double, MPFR.
- The supported environment is that C99 fesetround can be used. We can also use some options which enable
  - SSE2 of X86 CPU
  - emulation of rounded arithmetic using only nearest rounding
  - AVX-512 of newest Intel CPU
  - FMA instruction

Sample program of interval arithmetic

```
#include <kv/interval.hpp> // interval arithmeric
#include <kv/rdouble.hpp> // define rounded arithmetic for double

int main()
{
    kv::interval<double> s, x;

    std::cout.precision(17);

    s = 0;
    for (int i=1; i<=1000; i++) {
        x = i;
        s += 1/x;
    }

    std::cout << s << "\n";
}
```

```
[7.485470860549956,7.4854708605508238]
```

How to use

extract

```
$ ls
kv-0.4.48.tar.gz
$ tar xfz kv-0.4.48.tar.gz
$ ls
kv-0.4.48/  kv-0.4.48.tar.gz
$ cd kv-0.4.48
$ ls
LICENSE.txt  README.txt  example  kv  test
```

install

Place the kv directory under the favorite directory.
(e.g. `/usr/local/include/`)

compile & run

```
$ ls
interval.cc  kv/
$ c++ -I. -O3 interval.cc
$ ./a.out
[7.4854708605499956,7.4854708605508238]
```

sample program of double-double interval arithmetic

```
#include <kv/interval.hpp> // interval arithmeric
#include <kv/dd.hpp> // double-double
#include <kv/rdd.hpp> // define rounded arithmetic for dd

int main()
{
    kv::interval<kv::dd> s, x;

    std::cout.precision(34);

    s = 0;
    for (int i=1; i<=1000; i++) {
        x = i;
        s += 1/x;
    }

    std::cout << s << "\n";
}
```

```
[7.48547086055034491265651820433308257,7.48547086055034491265651820433360964]
```

double-double(dd)

- twosum: an algorithm to convert sum of two floating point numbers to the sum of two non-overlapping floating point numbers.
  (e.g. : $1234 + 5.432 \to 1239 + 0.432$)
- twoproduct: an algorithm to convert product of two floating point numbers to the sum of two non-overlapping floating point numbers.
  (e.g. : $1234 \times 5.432 \to 6703 + 0.088$)
- By using twosum and twoproduct, it is possible to realize a pseudo quadruple precision arithmeric using two double precision numbers.
- `dd.hpp` provides approximate calculation by double-double arithmetic.
- Using `dd.hpp` and `rdd.hpp` (define directed rounding for double-double arithmetic), it is possible to realize quadruple precision interval arithmetic.

mpfr

- a simple wrapper of the MPFR library to perform high precision floating point airhtmetic.
- `mpfr.hpp` provides approximate calculation by MPFR.
- e.g. `kv::mpfr<106>` means MPFR number with 106bit mantissa.
- Using `mpfr.hpp` and `rmpfr.hpp` (define directed rounding for MPFR), it is possible to realize interval arithmetic with MPFR. However, in the kv libary, only addition, subtraction, multiplication, division and square root of MPFR are used, and excellent mathematical functions of MPFR are not used.

Policy of implementing interaval arithmetic

- In IEEE 754 standard, only addition, subtraction, multiplication, divison and square root can trust and can change the rounding direction.
- We implemented interval arithmetic using only addition, subtraction, multiplication, division and square root.
- Verified interval mathematical functions are also implemented using only addition, subtraction, multiplication, division and square root.
- If addition, subtraction, multiplication, division and square root with directed rounding are implemented for some numeric type, we can use the number instead of "double". That is why dd and mpfr can be used for the endpoints of interval.

区間演算の内部型を切り替える仕組み(1)

- 現在のところ、`interval<double>`、`interval<dd>`、`interval<mpfr<N>>`の 3 種類の区間演算がサポートされている。
- `double`、`dd`、`mpfr<N>`それぞれで、方向付き丸めのやり方は異なる。未知の型の方向付き丸めはどうする?
- 加算、減算、乗算、除算、平方根のそれぞれについて、上向き丸め、下向き丸めで行なう関数 (10 種類) を作成し、クラスの特殊化によって差し替えることによって実現する。
- 数学関数は、この 10 種類の演算のみを用いて実装する。従って、端点に高精度な型を用いれば (その型に応じた) 高精度な数学関数が自動的に得られる。
- 特殊化を何もしないと、方向付き丸めを使わない (= 精度保証付き数値計算には使えない) 区間演算になる。
- `double`、`dd`、`mpfr<N>`それぞれに対する特殊化を行なう方法 (の一例) が、`rdouble.hpp`、`rdd.hpp`、`rmpfr.hpp`。

区間演算の内部型を切り替える仕組み (2)

interval.hpp 抜粋

```
template <class T> struct rop {
    static T add_up(const T& x, const T& y) {
        return x + y;
    }
    static T add_down(const T& x, const T& y) {
        return x + y;
    }
    static T sub_up(const T& x, const T& y) {
        return x - y;
    }
    static T sub_down(const T& x, const T& y) {
        return x - y;
    }
    static T mul_up(const T& x, const T& y) {
        return x * y;
    }
    static T mul_down(const T& x, const T& y) {
        return x * y;
    }
    static T div_up(const T& x, const T& y) {
        return x / y;
    }
    static T div_down(const T& x, const T& y) {
        return x / y;
    }
    static T sqrt_up(const T& x) {
        return sqrt(x);
    }
    static T sqrt_down(const T& x) {
        return sqrt(x);
    }
};

template <class T> class interval {
    T inf, sup;
    public:

    interval() {
        inf = 0.;
        sup = 0.;
    }
    template <class C> explicit interval(const C
        & x) {
        inf = x;
        sup = x;
    }
    template <class C1, class C2> interval(const
        C1& x, const C2& y) {
        inf = x;
        sup = y;
    }

    friend interval operator+(const interval& x,
        const interval& y) {
        interval r;
        r.inf = rop<T>::add_down(x.inf, y.inf);
        r.sup = rop<T>::add_up(x.sup, y.sup);
        return r;
    }

    template <class C> friend interval operator
      +(const interval& x, const C& y) {
        interval r;
        r.inf = rop<T>::add_down(x.inf, T(y));
        r.sup = rop<T>::add_up(x.sup, T(y));
        return r;
    }

    template <class C> friend interval operator
      +(const C& x, const interval& y) {
        interval r;
        r.inf = rop<T>::add_down(T(x), y.inf);
        r.sup = rop<T>::add_up(T(x), y.sup);
        return r;
    }

    friend interval operator-(const interval& x,
        const interval& y) {
        interval r;
        r.inf = rop<T>::sub_down(x.inf, y.sup);
        r.sup = rop<T>::sub_up(x.sup, y.inf);
        return r;
    }
};
```

区間演算の内部型を切り替える仕組み(3)

rdouble.hpp 抜粋

```
template <> struct rop <double> {
    static double add_up(const double& x, const
         double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_UPWARD);
        r = x1 + y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double add_down(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = -y;
        fesetround(FE_DOWNWARD);
        r = x1 + y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double sub_up(const double& x, const
         double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_UPWARD);
        r = x1 - y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double sub_down(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = -y;
        fesetround(FE_DOWNWARD);
        r = x1 - y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double mul_up(const double& x, const
         double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_UPWARD);
        r = x1 * y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double mul_down(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_DOWNWARD);
        r = x1 * y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double div_up(const double& x, const
         double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_UPWARD);
        r = x1 / y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double div_down(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y;
        fesetround(FE_DOWNWARD);
        r = x1 / y1;
        fesetround(FE_TONEAREST);
        return r;
    }
    static double sqrt_up(const double& x) {
        volatile double r, x1 = x;
        fesetround(FE_UPWARD);
        r = sqrt(x1);
        fesetround(FE_TONEAREST);
        return r;
    }
    static double sqrt_down(const double& x) {
        volatile double r, x1 = x;
        fesetround(FE_DOWNWARD);
        r = sqrt(x1);
        fesetround(FE_TONEAREST);
        return r;
    }
};
```

verification of nonlinear equations by Krawczyk's method

```
#include <kv/kraw-approx.hpp>
namespace ub = boost::numeric::ublas; // for abbreviation
struct Func { // define function object of f
    template <class T> ub::vector<T> operator() (const ub::vector<T>& x) {
        ub::vector<T> y(2);
        y(0) = x(0) * x(0) - x(1) - 1.;
        y(1) = (x(0) - 2.) * (x(0) - 2.) - x(1) - 1.;
        return y;
    }
};
int main() {
    ub::vector<double> x;
    ub::vector< kv::interval<double> > ix;
    std::cout.precision(17);
    x.resize(2);
    x(0) = 1.01; x(1) = 0.01; // initial value of Newton's method
    kv::krawczyk_approx(Func(), x, ix, 3, 1); // after 3 Newton iteration, check existence of the
         true solution near the approximate solution.
}
```

```
newton0: [2]([1,1],[-9.9999999999853679e-05,-9.9999999999853678e-05])
newton1: [2]([1,1],[2.4286128663675299e-17,2.42861286636753e-17])
newton2: [2]([1,1],[-3.1225022567582528e-17,-3.1225022567582527e-17])
I: [2]([0.99999999999999911,1.0000000000000009],[-3.9204750557075841e-16,3.2959746043559335e-16])
K: [2]([0.99999999999999977,1.0000000000000005],[-3.1225022567584106e-17,1.9081958235745036e-16])
```

finding all solutions of nonlinear equations

```
#include <kv/allsol.hpp>
namespace ub = boost::numeric::ublas;
struct Func { // define function object of f
    template <class T> ub::vector<T> operator() (const ub::vector<T>& x) {
        ub::vector<T> y(2);
        y(0) = x(0) * x(1) - cos(x(1));
        y(1) = x(0) - x(1) + 1;
        return y;
    }
};
int main()
{
    ub::vector< kv::interval<double> > x(2);
    std::cout.precision(17);
    x(0) = kv::interval<double>(-1000, 1000);
    x(1) = kv::interval<double>(-1000, 1000);
    kv::allsol(Func(), x); // find all solutions
}
```

```
[2]([-1.964111328125,-1.47607421875],[-0.66169175448117435,-0.47607421875])(ex)
[2]([-1.5500093499272621,-1.5500093499272609],[-0.55000934992726192,-0.55000934992726113])(ex:
     improved)
[2]([-0.011962890625,0.47607421875],[0.988037109375,1.47607421875])(ex)
[2]([0.25115183522207645,0.25115183522076507],[1.2511518352207642,1.2511518352207654])(ex:
     improved)
ne_test: 49, ex_test: 3, ne: 23, ex: 2, giveup: 0
```

automatic differentiation(autodif)

- provide Bottom-Up type (forward mode) automatic differentiation.

```
#include <kv/autodif.hpp>
namespace ub = boost::numeric::ublas;
// definition of function
template <class T> ub::vector<T> func(const
      ub::vector<T>& x) {
    ub::vector<T> y(2);
    y(0) = 2. * x(0) * x(0) * x(1) - 1.;
    y(1) = x(0) + 0.5 * x(1) * x(1) - 2.;
    return y;
}
int main()
{
    ub::vector<double> v1, v2;
    ub::vector< kv::autodif<double> > va1,
         va2;
    ub::matrix<double> m;
    v1.resize(2);
    v1(0) = 5.; v1(1) = 6.;
    // initialization
    va1 = kv::autodif<double>::init(v1);
    // call function
    va2 = func(va1);
    // split the result
    kv::autodif<double>::split(va2, v2, m);
    // f(5, 6)
    std::cout << v2 << "\n";
    // Jacobian matrix at (5, 6)
    std::cout << m << "\n";
}
```

```
[2](299,21)
[2,2]((120,50),(1,6))
```

Affine Arithmetic (`affine`)

Affine Arithmetic

- Affine arithmeric can supress overestimation of interval arithmetic. Instead it takes more computation time.
- Since dependency information with respect to other variables is held, overestimation can be supressed.
- All variables are expressed in affine form like $x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$. ε_i are dummy variables which satisfy $-1 \le \varepsilon_i \le 1$ and the coefficients x_i have dependency information.
- Every time a nonlinear operation such as multiplication, division or mathematical function appears, the number of dummy variables increases and the calculation speed slows down.

- Like as the `interval`, we can use `dd` or `mpfr` as well as `double` for internal type of `affine`.
- has a function to reduce the number of dummy variables.

Example of Affine Arithmetic

The QRT (Quispel-Roberts-Thompson) map

- calculate recurrence formula: $x_{n+1} = \dfrac{1 + \alpha x_n}{x_{n-1} x_n^{\sigma}}$
- with $\sigma = 2$, $\alpha = 2$, $x_0 = x_1 = 1$ by interval arithmeric and affine arithmetic.

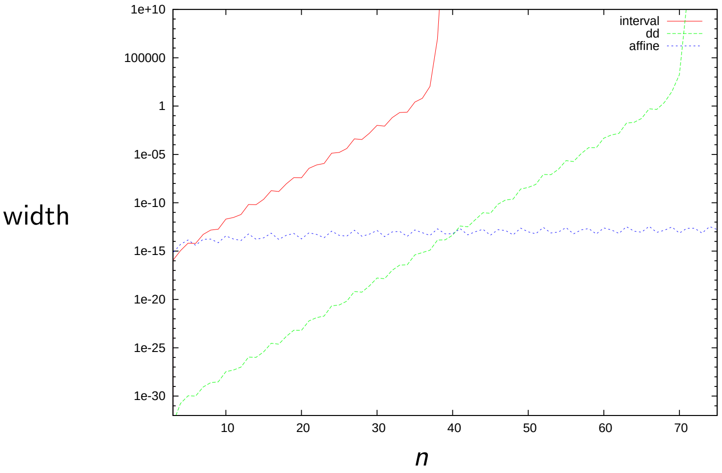
```
#include <kv/interval.hpp>
#include <kv/rdouble.hpp>
int main()
{
    int i;
    kv::interval<double> x, y, z;
    std::cout.precision(17);
    x = 1.;
    y = 1.;
    for (i=2; i<=10000; i++) {
        z = (1 + 2 * y) / (x * y * y);
        std::cout << i << " " << z << "\n";
        x = y;
        y = z;
    }
}
```

```
#include <kv/affine.hpp>
int main()
{
    int i;
    kv::affine<double> x, y, z;
    std::cout.precision(17);
    x = 1.;
    y = 1.;
    for (i=2; i<=10000; i++) {
        z = (1 + 2 * y) / (x * y * y);
        std::cout << i << " " << to_interval(
            z) << "\n";
        x = y;
        y = z;
    }
}
```

Results

n	interval arithmetic	affine arithmetic
2	[3, 3]	[3, 3]
3	[0.7777777777777767, 0.7777777777777778]	[0.7777777777777756, 0.7777777777777824]
4	[1.4081632653061222, 1.4081632653061232]	[1.4081632653061197, 1.4081632653061247]
5	[2.474480151228730 2, 2.4744801512287369]	[2.4744801512287271, 2.4744801512287414]
6	[0.6899539592210210 9, 0.6899539592210273 2]	[0.6899539592210222, 0.6899539592210262 1]
7	[2.020393474742363, 2.0203934747424169]	[2.0203934747423817, 2.0203934747423987]
8	[1.7898074714307314, 1.7898074714308816]	[1.7898074714307978, 1.7898074714308153]
⋮	⋮	⋮
31	[0.7009891618227720 4, 0.7094198209793560 8]	[0.7051917561686529 2, 0.7051917561686842 4]
32	[1.7816188152293368, 1.8444838202503787]	[1.8127715215496742, 1.8127715215497711]
33	[1.890688867011997, 2.1073484458445711]	[1.9960405520559754, 1.9960405520560838]
34	[0.5837212479498864 4, 0.8187930456850460 8]	[0.6911938131215669 1, 0.6911938131216002 3]
35	[1.5341327531940911, 4.0942614522583929]	[2.4982930525184534, 2.4982930525186054]
36	[0.2964039576199632 9, 6.6882779916662063]	[1.3900085495715059, 1.390008549571586]
37	[0.0086967943592607538, 106.66548725824453]	[0.7830967853484550 6, 0.7830967853484963 7]
38	[$1.3369859317919986 \times 10^{-5}$, 9560542.5436595381]	[3.0105168251706007, 3.0105168251708015]
39	[$1.0257053348148149 \times 10^{-16}$, $1.229984619387229 \times 10^{19}$]	[0.9892441618081190 2, 0.9892441618081792 1]
40	[$6.9138205018986439 \times 10^{-46}$, $1.7488703313159241 \times 10^{56}$]	[1.0109923081577889, 1.0109923081578503]
41	[$2.6581843623384974 \times 10^{-132}$, $7.1339291414655989 \times 10^{162}$]	[2.9887738208443805, 2.9887738208445911]
42	[0, ∞]	[0.7726251804826496, 0.7726251804826975 8]
43	–	[1.4265918581977079, 1.4265918581978075]
⋮	⋮	⋮
9999	–	[0.7607151065993281 7, 0.7607151066789953 4]
10000	–	[1.4727965248961243, 1.4727965251850226]

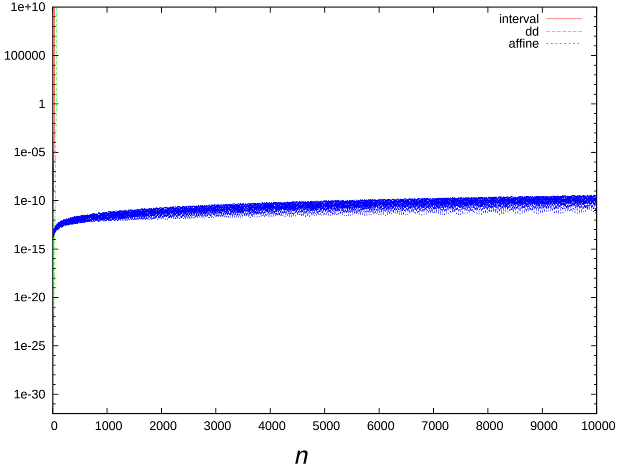
Results



(dd is the result by double-double (pseudo-quadruple precision) interval.)

Results ($n \leq 10000$)

width



Power Series Arithmetic (PSA)

- PSA can be used for the verified algorithms of solving ordinary differential equations, numerical integration, calculation of higher derivative, and so on.
- Two types of PSA. n: fixed integer

Type-I PSA	simply discard the terms higher than n.
Type-II PSA	include the influence of the terms higher than n into the interval coefficient of t^n.

Example of PSA (mutiplication)

$(1+2t-3t^2)\times(1-t+t^2)$	
Type-I PSA	Type-II PSA
not necessary to decide domain	domain $=[0,0.1]$
$1+t-4t^2$	$1+t+[-4,-3.5]t^2$

$$(1+2t-3t^2)(1-t+t^2)=1+t-4t^2+5t^3-3t^4$$
$$=1+t+(-4+5t-3t^2)t^2\in 1+t+[-4,-3.5]t^2$$

Type-I PSA

Power Series

$$x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n$$

- four basic operations $+, -, \times, \div$ between power series.
- mathematical functions $(\exp, \log,$ etc and $\int)$ for power series.
- leave the $(\leq n)$-th terms of result and discard the terms higher than n.
- almost same as:
    - Mathematica's "`Series`".
    - Intlab's "`taylor`".
    - automatic differentiation for higher order derivative.

Type-II PSA

Power Series

$$x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n$$

- operations are defined on fixed finite closed set $D = [t_1, t_2] \ni 0$.
- include the influence of the terms higher than n into the interval coefficient of t^n.
- all coefficients $x_0, \cdots, x_n$ are interval.
- however, in a typical case, $x_0, \cdots, x_{n-1}$ are narrow intervals and x_n becomes wide interval.

Operation Rules of Type-II PSA (1/5)

$$x(t) = x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n$$
$$y(t) = y_0 + y_1 t + y_2 t^2 + \cdots + y_n t^n$$

addition and subtraction

$$x(t) \pm y(t) = (x_0 \pm y_0) + (x_1 \pm y_1)t + \cdots (x_n \pm y_n)t^n$$

example of addition

$$x(t) = 1 + 2t - 3t^2$$
$$y(t) = 1 - t + t^2$$

$$x(t) + y(t) = 2 + t - 2t^2$$

Operation Rules of Type-II PSA (2/5)

multiplication

1. multiplication with no truncation

$$x(t) \times y(t) = z_0 + z_1 t + \cdots + z_{2n} t^{2n}$$

$$z_k = \sum_{i=\max(0,k-n)}^{\min(k,n)} x_i y_{k-i}$$

2. Order Reduction from $2n$ to n.

Definition: Order Reduction from m to n

$$x_0 + x_1 t + x_2 t^2 + \cdots + x_m t^m \Longrightarrow z_0 + z_1 t + \cdots + z_n t^n$$

$$z_i = x_i \quad (0 \le i \le n-1)$$

$$z_n = \left\{ \sum_{i=n}^{m} x_i t^{i-n} \mid t \in D \right\}$$

Operation Rules of Type-II PSA (3/5)

example of multiplication

Set domain $D = [0, 0.1]$.

$$x(t) = 1 + 2t - 3t^2$$
$$y(t) = 1 - t + t^2$$

$$\begin{aligned} x(t) \times y(t) &= 1 + t - 4t^2 + 5t^3 - 3t^4 \\ &= 1 + t + (-4 + 5t - 3t^2)t^2 \\ &\in 1 + t + \left\{-4 + 5t - 3t^2 \mid t \in [0, 0.1]\right\} t^2 \\ &= 1 + t + [-4, -3.5]t^2 \end{aligned}$$

Operation Rules of Type-II PSA (4/5)

mathematical functions such as sin, etc

For function g, substitute input in the Taylor expansion of g at x_0:

$$
\begin{aligned}
&g(x_0 + x_1 t + \cdots + x_n t^n) \\
&= g(x_0) + \sum_{i=1}^{n-1} \frac{1}{i!} g^{(i)}(x_0)(x_1 t + \cdots + x_n t^n)^i \\
&+ \frac{1}{n!} g^{(n)} \left(\mathrm{hull} \left(x_0, \left\{ \sum_{i=0}^{n} x_i t^i \;\middle|\; t \in D \right\} \right) \right) (x_1 t + \cdots + x_n t^n)^n
\end{aligned}
$$

All additions and multiplications in above calculation are executed by the type-II PSA.

Operation Rules of Type-II PSA (5/5)

Division

$x \div y = x \times (1/y)$ (reciprocal function and mulplication)

Indefinit Integral

$$\int_0^t x(t)dt = x_0 t + \frac{x_1}{2} t^2 + \cdots \frac{x_n}{n+1} t^{n+1}$$

Verified Numerical Quadrature

Numerical quadrature on interval $[x_i, x_i + \Delta t]$

$$\int_{x_i}^{x_i+\Delta t} f(t)dt$$

is as follows:

1. Using order-n power series

   $$x(t) = 0 + t \quad (+0t^2 + \cdots 0t^n) \quad ,$$

   Calculate

   $$y(t) = \int_0^t f(x_i + x(t))dt$$

   by type-II PSA with domain $[0, \Delta t]$.
2. Let the calculation result $y(t)$ be

   $$y(t) = y_1 t + y_2 t^2 + \cdots y_{n+1} t^{n+1},$$

   the integral value is obtained by $y(\Delta t)$.

How to control step size

ε_0: expected local error (e.g. machine epsilon)

1. Calculate Taylor expansion of the solution by Type-I PSA, and estimate appropriate step size Δt_0 using coefficients of the Taylor expansion. For Taylor expansion

$$x_0 + x_1 t + x_2 t^2 + \cdots + x_{n-1} t^{n-1} + x_n t^n \,,$$

   estimate step size as:

$$\Delta t_0 = \frac{\varepsilon_0^{\frac{1}{n}}}{\max(\|x_{n-1}\|^{\frac{1}{n-1}}, \|x_n\|^{\frac{1}{n}})} \,.$$

2. calculate verified solution by Type-II PSA using the step size Δt_0 .
3. Using ε, which is the real error of the above verified solution, estimate new step size Δt_1 as:

$$\Delta t_1 = \Delta t_0 \left(\frac{\varepsilon_0}{\varepsilon} \right)^{\frac{1}{n}}$$

4. calculate verified solution by Type-II PSA using the step size Δt_1 .

Example of Verified Numerical Quadrature (1/3)

$$\int_0^{10} \frac{\sin(x)}{\cos(x^2) + 1 + 2^{-10}} dx$$

Program

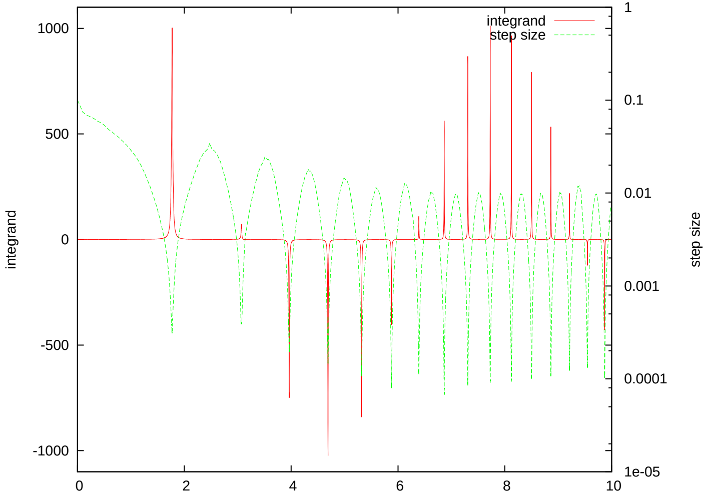
```
#include <iostream>
#include <kv/defint.hpp>

typedef kv::interval<double> itv;

struct Func {
    template <class T> T operator() (const T& x) {
        return sin(x) / (cos(x*x) + 1. + pow(2., -10));
    }
};

int main() {
    std::cout.precision(17);

    std::cout << kv::defint_autostep(Func(), (itv)0., (itv)10., 10) << "\n";
}
```

Example of Verified Numerical Quadrature (2/3)

$$\int_0^{10} \frac{\sin(x)}{\cos(x^2) + 1 + 2^{-10}} dx$$

kv-0.4.41	[38.383526264535227,38.383526264964969]
intlab 9	[38.34845927756175, 38.41859325162576]
octave 3.8.1	38.3837105761501
Mathematica 10.1.0	0.0608979
matlab 2007b	38.383519835854528
keisan (Romberg)	38.324147930794
keisan (Tanh-Sinh)	38.24858948837754677984
keisan (Gauss-Legendre)	116.448156707725851273
intde2 by ooura	32.4641
python + scipy	36.48985372847387
CASIO fx-5800P	38.38352669

Example of Verified Numerical Quadrature (3/3)



Solving Initial Value Problem of ODEs

Problem

$$\frac{dx}{dt} = f(x,t), \quad x \in \mathbb{R}^l, t \in \mathbb{R}$$
$$x(t_0) = x_0$$

Our Algorithm for Initial Value Problems

For $t_0 < t_1 < t_2 < \ldots,$

- Power Series Arithmetic (PSA) based algorithm to calculate verified value of $x(t_{i+1})$ based on $x(t_i)$. (1-step algorithm: verification algorithm for small step size.)
- Affine Arithmetic based algorithm to connect the solutions of 1-step algorithm over long time while supressing inflation of interval width.

Overview of 1-step Method (1/3)

Consider calculcating the value of $x(t_e)$ based on the initial value $v = x(t_s)$.

Origin Shift and Picard's fixed point method

$$x(t) = v + \int_0^t f(x(t), t + t_s)dt$$
$$(v = x(t_s), \quad t \in [0, t_e - t_s])$$

Generating Taylor Expansion of the solution

Set power series variable $X_0 = v$, $T = t$ and set $k = 0$

1. calculate the following by Type-I PSA with order k:

$$X_{k+1} = v + \int_0^t f(X_k, T + t_s)dt$$

2. $k = k + 1$.

repeat above procedure n times, then we can generate the order n Taylor expansion of the solutions as X_n.

Overview of 1-step Method (2/3)

Verification of Existence of the solution

Set domain $D = [0, t_{i+1} - t_i]$ for type-II PSA, using the following order n Taylor expansion generated by Type-I PSA:

$$X_n = x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n$$

and $T = t$,

1. Make candidate set Y_c:

   $$Y_c = x_0 + x_1 t + x_2 t^2 + \cdots + V_c t^n$$

   by inflating the coefficient of last term of X_n.
2. Calculate $v + \int_0^t f(Y_c, T + t_s)dt$ by Type-II PSA with order n and reduce the order from $n+1$ to n:

   $$Y = x_0 + x_1 t + x_2 t^2 + \cdots + V t^n$$

   Notice that the coefficients of Y upto order $n-1$ become completely equal to that of X_n.
3. if $V \subset V_c$ then the solution exists in Y.

Overview of 1-step Method (3/3)

For example, we can make the candidate set which is expected to include solution as follows:

Making Candidate Set

1. Cauculate $v + \int_0^t f(X_n, T + t_s)dt$ by Type-II PSA with order n and reduce the order from $n+1$ to n : $Y_0 = x_0 + x_1 t + \cdots + V_0 t^n$
2. Set $r = ||V_0 - x_n||$ and then candidate set V is obtained as:

$$V_c = x_n + 2r\,([-1,1],\ldots,[-1,1])^T$$

(Let radius be twice of the Newton-like step.)

Example of 1-step Method

$$\frac{dx}{dt} = -x^2$$
$$x(0) = 1, \quad t \in [0, 0.1]$$

Order of Taylor Expansion: $n = 2$, use 3-digit decimal number.

(generating Taylor expansion by Type-I PSA)

$$\begin{aligned}
X_0 &= \boxed{1} \\
X_1 &= 1 + \int_0^t (-X_0^2)dt \\
&= 1 + \int_0^t (-1)dt \\
&= \boxed{1 - t} \\
X_2 &= 1 + \int_0^t (-X_1^2)dt \\
&= 1 + \int_0^t (-(1-t)^2)dt \\
&= 1 + \int_0^t (-(1-2t))dt \\
&= \boxed{1 - t + t^2}
\end{aligned}$$

(generating candidate set)

$$\begin{aligned}
&1 + \int_0^t (-X_2^2)dt \\
&= 1 + \int_0^t (-(1-t+t^2)^2)dt \\
&= 1 + \int_0^t (-(1-2t+[2.8,3]t^2))dt \\
&= 1 - t + t^2 + [-1, -0.933]t^3
\end{aligned}$$

reduce the order from 3 to 2:

$$Y_0 = 1 - t + [0.9, 1]t^2$$

because $r = ||[0.9, 1] - 1|| = 0.1$,

$$Y_c = \boxed{1 - t + [0.8, 1.2]t^2}$$

(verification of solution by Type-II PSA)

$$\begin{aligned}
&1 + \int_0^t (-Y_c^2)dt \\
&= 1 - t + t^2 + [-1.14, -0.786]t^3
\end{aligned}$$

reduce the order from 3 to 2:

$$Y = \boxed{1 - t + [0.886, 1]t^2}$$

because $[0.886, 1] \subset [0.8, 1.2]$, true

solution exists in Y.

Connecting 1-step method over long time (1/2)

Flow Map

For ODEs, flow map is the map which maps the value $x(t_s)$ (initial value at $t = t_s$) to the value $x(t_e)$ (value of solution at $t = t_e$).

$$\phi_{t_s,t_e} : \mathbb{R}^l \to \mathbb{R}^l, \quad \phi_{t_s,t_e} : x(t_s) \mapsto x(t_e)$$

Variational Equation with respect to Initial Value

Let $x^*(t)$ be the solution of ODE with inital value v, by solving matrix ODE:

$$\frac{d}{dt}y(t) = f_x(x^*(t), t)y(t), \quad y \in \mathbb{R}^{l \times l}$$
$$y(t_s) = I, \quad t \in [t_s, t_e]$$

we can obtain Jacobian of flow map by $\phi'_{t_s,t_e}(v) = y(t_e)$

Connecting 1-step method over long time (2/2)

Let J_i be an inclusion of the true solution at t_i.

Mean Value Form

$$\phi_{t_i,t_{i+1}}(x) \in \phi_{t_i,t_{i+1}}(\mathrm{mid}(J_i)) + \phi'_{t_i,t_{i+1}}(J_i)(x - \mathrm{mid}(J_i))$$

- Direct calculation may cause the inflation of the interval width by wrapping effect.
- Use affine arithmetic to supress wrapping effect.

How to control step size

ε_0: expected local error (e.g. machine epsilon)

1. Calculate Taylor expansion of the solution by Type-I PSA, and estimate appropriate step size Δt_0 using coefficients of the Taylor expansion. For Taylor expansion

$$x_0 + x_1 t + x_2 t^2 + \cdots + x_{n-1} t^{n-1} + x_n t^n ,$$

estimate step size as:

$$\Delta t_0 = \frac{\varepsilon_0^{\frac{1}{n}}}{\max(\|x_{n-1}\|^{\frac{1}{n-1}}, \|x_n\|^{\frac{1}{n}})} .$$

2. calculate candidate set by Type-II PSA using the step size Δt_0 .
3. Using ε, which is the newly mixed error when we use the candidate set, estimate new step size Δt_1 as:

$$\Delta t_1 = \Delta t_0 \left(\frac{\varepsilon_0}{\varepsilon} \right)^{\frac{1}{n}}$$

4. calculate candidate set by Type-II PSA using the step size Δt_1, and verify existence of the true solution. If the verification fails, for example, halve the step size.

Example: initial value problem (1/4)

van del Pol equation

$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = 0$$

change to first order system

$$\begin{aligned}\frac{dx}{dt} &= y\\ \frac{dy}{dt} &= \mu(1-x^2)y - x\end{aligned}$$

Example: initial value problem (2/4)

```
#include <kv/ode-maffine.hpp>
namespace ub = boost::numeric::ublas;
typedef kv::interval<double> itv;
class VDP { // define function object for right hand side of ODE
    public:
    template <class T> ub::vector<T> operator() (const ub::vector<T>& x, T t){
        ub::vector<T> y(2);
        y(0) = x(1);
        y(1) = 1. * (1. - x(0)*x(0)) * x(1) - x(0);
        return y;
    }
};

int main()
{
    ub::vector<itv> x;
    itv end;
    std::cout.precision(17);
    x.resize(2);
    x(0) = 1.; // initial value
    x(1) = 1.;
    end = 100.; // end time
    kv::odelong_maffine(VDP(), x, itv(0.), end); // solve initial value problem (from 0 to end)
    std::cout << x << "\n";
}
```

```
[2]([2.007790480952114,2.007790480952139],[-0.056051438751153989,-0.056051438750559116])
```

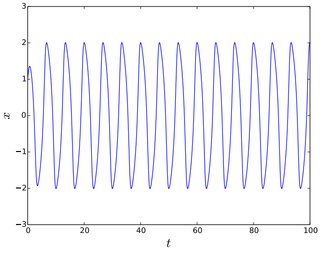
Example: initial value problem (3/4) (dense output)

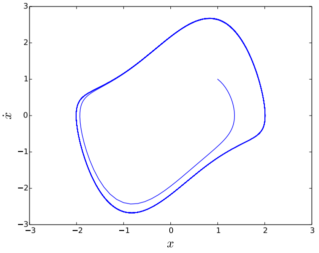
dense output with stepsize 2^{-4}.

```
#include <kv/ode-maffine.hpp>
#include <kv/ode-callback.hpp>
namespace ub = boost::numeric::ublas;
typedef kv::interval<double> itv;
class VDP {
    public:
    template <class T> ub::vector<T> operator() (const ub::vector<T>& x, T t){
        ub::vector<T> y(2);
        y(0) = x(1);
        y(1) = 1. * (1. - x(0)*x(0)) * x(1) - x(0);
        return y;
    }
};

int main()
{
    ub::vector<itv> x;
    itv end;
    std::cout.precision(17);
    x.resize(2);
    x(0) = 1.;
    x(1) = 1;
    end = 100.;
    kv::odelong_maffine(VDP(), x, itv(0.), end, kv::ode_param<double>(), kv::
        ode_callback_dense_print<double>(itv(0.), itv(pow(2., -4))));
}
```

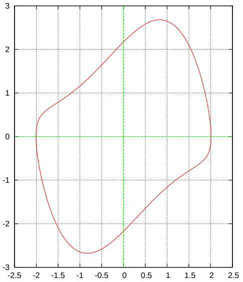
Example: initial value problem (4/4) (dense output)

```
t: [-0,0]
[2]([1,1],[1,1])
t: [0.0625,0.0625]
[2]([1.0604282381493324,1.0604282381493327],[0.93186430539999509,0.93186430539999521])
t: [0.125,0.125]
[2]([1.1162696582692208,1.1162696582692211],[0.8534995323034189,0.8534995323034190 2])
t: [0.1875,0.1875]
[2]([1.1669406889500346,1.1669406889500352],[0.76674349796008578,0.766743497960085 9])
t: [0.25,0.25]
[2]([1.211981145751376,1.2119811457513768],[0.67368071112755956,0.6736807111275599])
----------------omitted----------------
t: [99.9375,99.9375]
[2]([2.0074651477352984,2.0074651477354078],[0.070452402413564 79,0.070452402414533405])
t: [100,100]
[2]([2.0077904809520377,2.007790480952215],[-0.0560514387514576,-0.05605143875025545])
```





Example: boundary value problem (1/2)



$$\begin{cases} \begin{pmatrix} x \\ y \end{pmatrix} - \phi_{0,p} \begin{pmatrix} x \\ y \end{pmatrix} = 0 \\ s \begin{pmatrix} x \\ y \end{pmatrix} = 0 \end{cases}$$

$\phi_{0,p}$: flow map from 0 to p

p : period

s : equation of Poincaré section

(x, y, p) : unknown variables

Verification of periodic solusion of van der Pol Equation ($\mu = 1$) using Krawczyk Method for Poincaré Map (Poincaré section: $x = 0$). The Verified period is

$$T \in [6.663286859323\mathbf{1044}, 6.663286859323\mathbf{1534}]$$

Example: boundary value problem (2/2)

```
#include <kv/poincaremap.hpp>
#include <kv/kraw-approx.hpp>
namespace ub = boost::numeric::ublas;
typedef kv::interval<double> itvd;
class VDP {
    public:
    template <class T> ub::vector<T> operator() (ub::vector<T> x, T t){
        ub::vector<T> y(2);
        y(0) = x(1);
        y(1) = 1. * (1. - x(0)*x(0)) * x(1) - x(0);
        return y;
    }
};
class VDPPoincareSection {
    public:
    template <class T> T operator() (ub::vector<T> x){
        T y;
        y = x(0) - 0.;
        return y;
    }
};
int main()
{
    ub::vector<double> x;
    ub::vector<itvd> ix;
    std::cout.precision(17);
    VDP f;
    VDPPoincareSection g;
    kv::PoincareMap<VDP, VDPPoincareSection, double> h(f, g, (itvd)0.);
    x.resize(3); x(0) = 0.; x(1) = 1.; x(2) = 6.28;
    kv::krawczyk_approx(h, x, ix, 10, 0);
    std::cout << ix << std::endl;
}
```

```
[3]([-5.4587345687103157e-30,5.458734568710315e
    -30],[2.1727136926224956,2.1727136926225979],[6.6632868593231044,6.6632868593231534])
```

非線形方程式の全解探索の実例 (1/2)

5 つの解を持つ 2-トランジスタ回路方程式

$$\begin{pmatrix} 1 & -0.5 & 0 & 0 \\ -0.99 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.5 \\ 0 & 0 & -0.99 & 1 \end{pmatrix} \begin{pmatrix} 10^{-9}/0.99 \times (\exp(x_1/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_2/0.053) - 1) \\ 10^{-9}/0.99 \times (\exp(x_3/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_4/0.053) - 1) \end{pmatrix}$$

$$+ 10^{-4} \begin{pmatrix} 4 & -3 & -2 & 1 \\ -3 & 3 & 1 & 0 \\ -2 & 1 & 4 & -3 \\ 1 & 0 & -3 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} -0.001 \\ 0.000936 \\ -0.001 \\ 0.000936 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$x_1, x_2, x_3, x_4 \in [-10, 10]$

Yusuke Nakaya, Tetsuo Nishi, Shin'ichi Oishi, and Martin Claus: "Numerical Existence Proof of Five Solutions for Certain Two-Transistor Circuit Equations", Japan J. Indust. Appl. Math. Volume 26, Number 2-3, pp.327–336, 2009

非線形方程式の全解探索の実例 (2/2)

- 2-トランジスタ回路は高々3 つの解 (直流動作点) しか持たないという conjecture が知られていたが、この論文はそれを否定する 5 つの解を持つ例を示した。
- kv に含まれる全解探索プログラム (`allsol.hpp`) を使用すると、295668 回の非存在テストと 145259 回の存在テストを実行し、147816 個の非存在領域と 5 つの存在領域の特定に成功した。

精度保証された解

	x_1	x_2
	x_3	x_4
(1)	[0.70358963169344701,0.70358963169362677]	[-0.72180712343566756,-0.72180712342886677]
	[0.74231647296775893,0.74231647296781967]	[0.61988728360925959,0.61988728360955603]
(2)	[0.71990164129087852,0.71990164129099532]	[-0.003335867014043125,-0.0033358670081318586]
	[0.73551253992991871,0.73551253992997967]	[0.57555293496204418,0.57555293496258942]
(3)	[0.74231647296775959,0.74231647296781911]	[0.61988728360926281,0.61988728360955337]
	[0.70358963169344879,0.70358963169362488]	[-0.72180712343560172,-0.7218071234289427]
(4)	[0.73551253992991894,0.73551253992997923]	[0.57555293496204606,0.57555293496258542]
	[0.7199016412908793,0.7199016412909951]	[-0.0033358670140009599,-0.0033358670081407915]
(5)	[0.72928963256368528,0.72928963256369895]	[0.47145516180306784,0.47145516180350878]
	[0.72928963256368528,0.72928963256369895]	[0.47145516180306701,0.47145516180350833]

kv の様々なコンパイルオプション

- (オプション無し) `fesetround` による丸め変更。C99 準拠の幅広い環境で使える。
- `-DKV_FASTROUND` SSE2 命令 (`_mm_setcsr`) による丸め変更。Intel x64 環境のみ。
- `-DKV_NOHWROUND` 丸めの変更を一切行わず、方向付き丸めを最近点丸めのみで完全にエミュレートする。IEEE754 準拠の幅広い環境で使えるが、遅い。
- `-DKV_USE_AVX512` AVX-512 命令を使う。Intel の最新の CPU のみ。
- `-DKV_USE_TPFMA` twoproduct に FMA 命令を使う。処理系の持つ `fma` 命令を使うが、正常に動作するか、高速化するかは環境次第。

非線形方程式の全解探索 (5 solution) での計算時間比較

計算環境

Intel Core i9 7900X 3.3GHz/4.3GHz, Memory 128G
Ubuntu 16.04 LTS
gcc 5.4.0

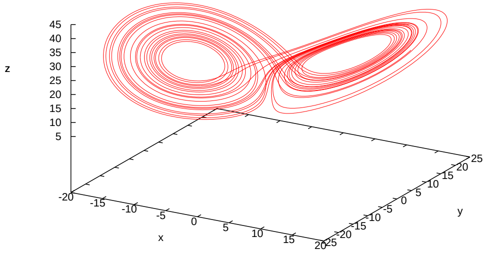
計算時間

計算精度	コンパイルオプション	計算時間	
		-DKV_USE_TPFMA なし	-DKV_USE_TPFMA あり
double	-O3	10.81 sec	
	-O3 -DKV_FASTROUND	8.40 sec	
	-O3 -DKV_NOHWROUND	19.91 sec	12.68 sec
	-O3 -DKV_USE_AVX512 -mavx512f	5.55 sec	
dd	-O3	54.10 sec	44.94 sec
	-O3 -DKV_FASTROUND	46.08 sec	36.72 sec
	-O3 -DKV_NOHWROUND	185.7 sec	118.9 sec
	-O3 -DKV_USE_AVX512 -mavx512f	21.42 sec	

ローレンツ方程式

$$\frac{dx}{dt} = 10(y - x)$$
$$\frac{dy}{dt} = 28x - y - xz$$
$$\frac{dz}{dt} = -\frac{8}{3}z + xy$$

$$(x(0), y(0), z(0)) = (36, 15, 15)$$

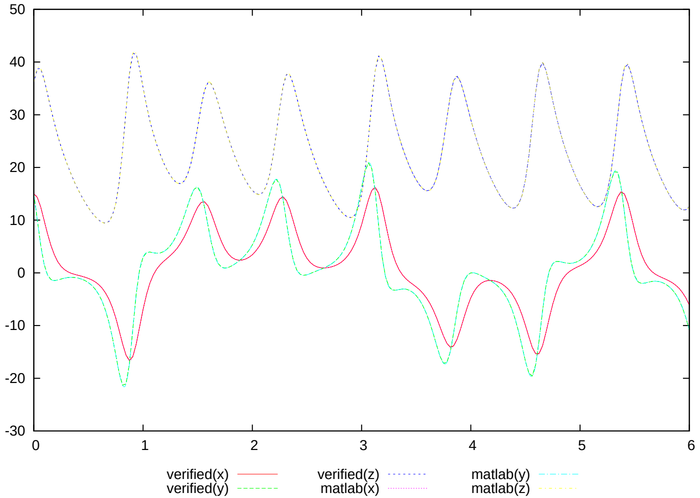


- カオス現象の発見で有名な方程式
- matlab という世界で最も信頼されている数値計算ソフトウェアと、自作の精度保証付き数値計算プログラムで解き比べてみる。

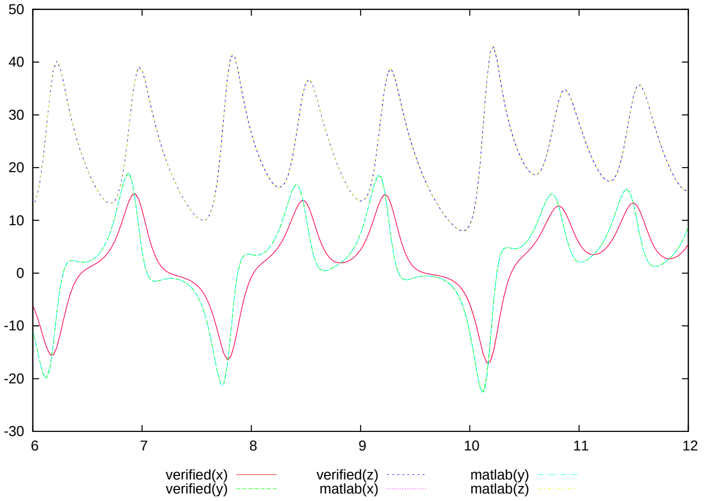
計算結果をグラフで見ると . . .

- matlab 側は ode45 に最高精度 (相対誤差 10^{-15}) を指定、精度保証側は kv ライブラリ。
- 横軸が時刻 t、縦軸は x, y, z の値。
- 最初はぴったり重なっているように見える。
- $t = 13$ あたりからわずかにずれが見えはじめ、$t = 15$ あたりで決定的に運命が分かれ、それ以降は異なる人生を歩んでいるかのよう。
- $t = 28$ 付近になると、精度保証付き数値計算プログラムの方のグラフに幅が出てくる。実はずっと上限と下限の 2 本のグラフを描画していた。このへんまでが double での限界。
- double-double を使えば $t = 50$ あたりまで行ける。

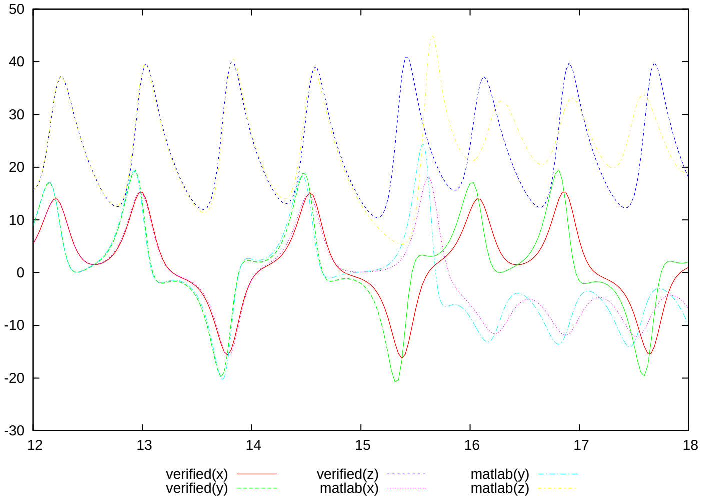
ローレンツ方程式 ($t = 6$ まで)



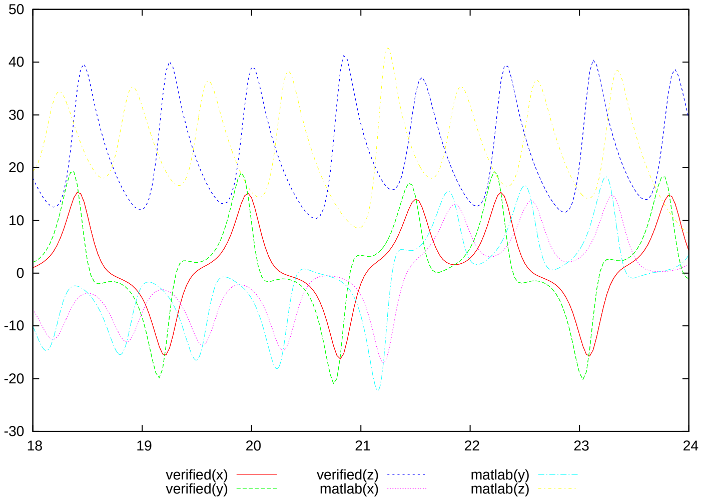
ローレンツ方程式 ($t = 12$ まで)



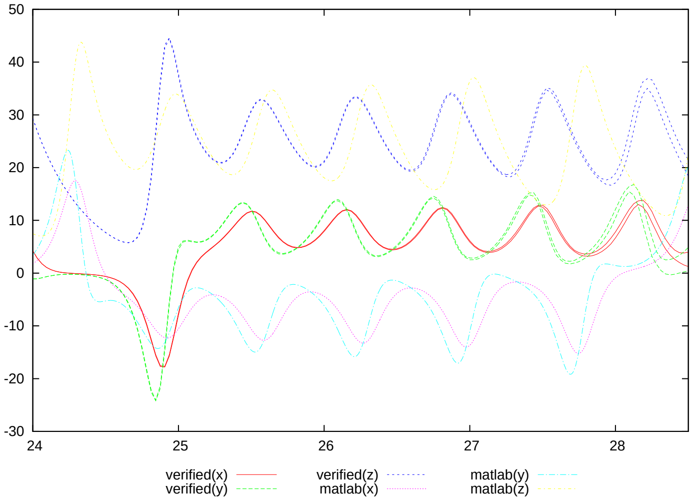
ローレンツ方程式 ($t = 18$ まで)



ローレンツ方程式 ($t = 24$ まで)



ローレンツ方程式 ($t = 28.5$ まで)



Conclusion

kv library

- Available to download at http://verifiedby.me/kv/.
- Written in C++. The boost C++ Library is required.
- Header-only. kv library is designed to work without "install" but only with the header files in itself.
- Open source. If we assert that the result of verified numerical computation is ‘proof’, all programs used for computation must be public.
- Data type of numbers in the calculation is not restrected to "double". Data type can be easily changed using "template" feature in C++.
- (Data Type) double, interval (with many verified mathematical functions), double-double, MPFR, complex, automatic differentiation, affine arithmeric, Power Series Arithmeric (PSA), and these combinations.
- (Applications) verified solution of nonlinear equations by Krawczyk method, finding all solutions of nonlinear equations, initial value problems of ODE, boundary value problems of ODE, etc.

We are waiting for the use of kv library for your research!