

Verified Numerical Computation and kv Library

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

Waseda University, Japan

(May 30, 2026)

- error in numerical computation
- verified numerical computation
- interval arithmetic
- Krawczyk method
- introduction of kv library
 - outline
 - interval arithmetic
 - automatic differentiation, double-double arithmetic, mpfr, affine arithmetic
 - 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

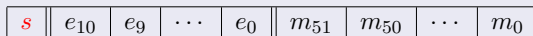
To solve "mathematical problems that can not be solved analytically" **by computer.**

- 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 developed 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 of radix)" such as 6.02×10^{23} .
- stored in binary numbers and use 64bits=8bytes.
- 64bits are divided as follows:



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)

-2^{1024} -2^{-1022} -2^{-1074} 2^{-1074} 2^{-1022} 2^{1024}

$-\infty$	normal	denorm	-0	$+0$	denorm	normal	∞
-----------	--------	--------	------	------	--------	--------	----------

Rounding Error

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

$$1 \div 3 = 0.3333333333333333148\dots$$

This is called **rounding error**.

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

$$1 \div 10 = 0.100000000000000005551\dots$$

- The error is not 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) $\frac{-b + \sqrt{b^2 - 4ac}}{2a} = -0.125$
- (rationalizing numerator) $\frac{2c}{-b - \sqrt{b^2 - 4ac}} = -0.100000000000000002$

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...

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

Rump's Example

Rump's Example

Calculate the following expression for $a = 77617$ $b = 33096$:

$$(333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + a/(2b)$$

Numerical Results

1.17260396480560302734375 (float)

1.1726039400531786949244406059733592 (double)

1.1726039400531786318588349045201801 (double-double)

1.1726039400531786318588349045201838 (mpfr113)

1.1726039400531786318588349045201838 (binary128)

-0.82739605994682136814116509547981629 (mpfr150) ← true value

From float (23bits) to binary128 (113bits), depending on the accuracy, it looks like it is calculated correctly, but it is actually wrong. We can see the true value with 150bits calculation.

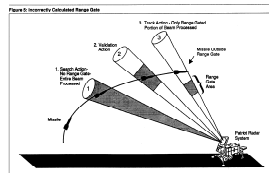
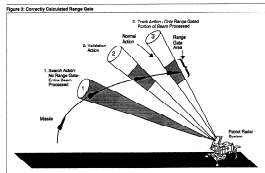
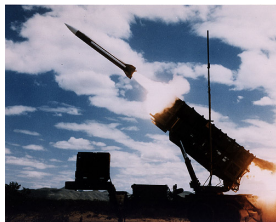
Patriot Missile Accident in the Gulf War

<http://www-users.math.umn.edu/~arnold/disasters/patriot.html>

<https://www.gao.gov/assets/220/215614.pdf>

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 related to the representation of "0.1 second", 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



Verified Numerical Computation

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

Required Technology

- **Interval Arithmetic** (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 (1/7)

Interval Arithmetic

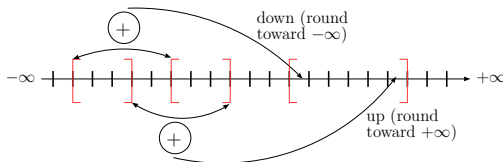
- Interval arithmetic is the most fundamental technique for verified numerical computation.
- For example, if $\pi \simeq 3.14$, it is **approximation**, but if $\pi \in [3.14, 3.15]$, it is **correct information**.
- In interval arithmetic, numerical values are expressed by **closed interval $X = [a, b]$ where the endpoints are floating point numbers that can be represented by a computer**.
- In interval arithmetic, the operation between intervals is performed so as to **include all possible calculation results for the values included in the interval operand**.
- In interval arithmetic, generally, **directed rounding** defined by IEEE 754 standard is used.
- There are two rolls of interval arithmetic, **estimation of rounding error** and **evaluation of range of function**.

Interval Arithmetic (2/7)

Rounding mode of IEEE754

- **round to nearest**: Default rounding mode. rounding to the nearest representable floating point number.
- **round toward $+\infty$** : rounding upward.
- **round toward $-\infty$** : rounding downward.
- **round toward 0**: chop rounding.

In the interval arithmetic, the lower bound is calculated by rounding downward and the upper bound is calculated by rounding upward, so that the calculation result inflates outward.



Interval Arithmetic (3/7)

- $X = [a, b], Y = [c, d]$
- $\underline{\cdot}$ 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** $X \times Y =$

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

- **Division** $X/Y =$

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

 (defined only when $Y \neq 0$)

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

Interval Arithmetic (4/7)

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

- Numerical Calculation (Not Verified)

$$1 \div 3 = 0.333$$

$$0.333 \times 3 = 0.999$$

Error due to rounding off is observed.

- 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) $\frac{-b + \sqrt{b^2 - 4ac}}{2a} = [-0.1875, -0.0625]$
- (rationalizing numerator) $\frac{-b + \sqrt{b^2 - 4ac}}{2c} = [-0.100000000000000004, -0.099999999999999991]$

Interval Arithmetic (5/7)

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
0.099999999999999991
0.10000000000000001
```

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. \implies 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 Arithmetic (7/7)

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 $[-1, -0.99]$.

Suppressing overestimation

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

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

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

Affine Arithmetic (later mentioned)

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

A Method to guarantee the existence of a solution of nonlinear equation

$$f(x) = 0, \quad f : \mathbb{R}^n \rightarrow \mathbb{R}^n .$$

Krawczyk method

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

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

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

$$f'(I) = \begin{pmatrix} [1.2, 1.6] & [1.2, 1.6] \\ 1 & -1 \end{pmatrix} \text{ and}$$

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

. This guarantees existence of unique solution.

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

Application of Krawczyk method

Guaranteeing the existence of the true solution based on the approximate solution c

For $R \simeq f'(c)^{-1}$ and $r = 2\|Rf(c)\|$ (twice of the modification of Newton method), use

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

as the candidate set for Krawczyk method.

Finding all solutions of nonlinear equations in interval vector I

Apply two theorems:

- **Existence theorem** in interval vector I (Krawczyk method)
- **Non-Existence theorem** in interval vector I (for example, if $f(I) \not\supseteq 0$ then there is no solution in I)

for interval I , and divide the interval recursively until one of the theorems holds.

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 0.4.60.
- 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 restricted 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 arithmetic
- Power Series Arithmetic (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

- 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 English)

English [1/16]

Last Update: Mar 14, 2024



kv - a C++ Library for Verified Numerical Computation

Masahito Kashiwagi

1. Introduction

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

For the version of verified numerical computation and for the library please see the table [kv-verified-computations](#) (2 pages)

All of programs of the library that have been distributed from the version 0.01 to the spring 2013, the current library is based. Discussion will be made in the general session of the library team. 3. Download and Install, all programs with respect to actual verifications are originally produced, whereas some functions in kv itself can partially work in the library. E.g. function files for some algorithms.

2. Requirement

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

We need a change compiler mode to make formal verifications. See [1. General Information](#) and [2. Change Compiler Mode and Compiler Options](#) for details of the requirements on CPU and compiler.

Actually, the program for kv is developed on Windows 11.0x (64bit) + gcc. This operating system (OS) are recommended, while it was confirmed that the library works on the following computational environments.

- ubuntu 22.04.04LTS + gcc 11.2.0
- ubuntu 22.04.04LTS + clang 14.0.0
- ubuntu 20.04.04LTS + gcc 7.5.0
- ubuntu 20.04.04LTS + clang 10.0.0
- ubuntu 18.04.04LTS + gcc 7.5.0
- ubuntu 18.04.04LTS + clang 6.0.0
- ubuntu 16.04.04LTS + gcc 5.3
- ubuntu 16.04.04LTS + clang 5.0
- ubuntu 14.04.04LTS + gcc 4.8
- ubuntu 14.04.04LTS + gcc
- ubuntu 10.04.04LTS + gcc
- windows7 (64bit) + Visual Studio 2012
- windows7 (64bit) + Visual Studio 2013
- windows7 (64bit) + Visual Studio 2008
- windows7 (64bit) + cygwin + gcc
- Windows7 (64bit) + MSVC9.0 + gcc + g++
- MS-DOS/Win 9x + Mac OS X/Linux + gcc
- Mac OS X/Linux + gcc
- ubuntu 22.04 on ARM64/AArch64
- raspberry pi 4 + cygwin + gcc
- Sony SWS/PS4 PS5 + ubuntu 18.04 + gcc
- Intel Edison + gcc
- iPhone 13 (ARM64) on darwin/arm64T802 + gcc

3. Download and Install

Download [kv-verified-computations](#) (updated on Mar. 14, 2024)
(384 versions are available to download [here](#).)

You can access to the source code of kv in [kv-verified-computations](#).

Since the library is designed to work "without" "compiler" and "runtime" "loader" but only with the linker files in kv itself. After expanding the archive file of the kv library, the three directories of "kv", "kvlib" and "kvmainlib" will be constructed. The main components of the kv library are in "kv" directory. Therefore, the kv library is available after setting the directory "kv" along with all the files in kv on an appropriate directory (e.g. the current directory or kv sub-directory).

The operation mode and functions provided by the library can be used in the counterpart of "kv" so that these functions also can cooperate with other libraries.

The operation mode of the library can be done by compiling some kv file to "kv" or "kvmainlib". For example, also

- expanding the archive file of kv library and
- setting kv on kv sub-directory.

you can verify the operation of kv library by executing the following command (see "kv" directory).

```
*** 1. -Dkvlibinclude test-interval.cc
```

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

It is recommended to specify the compiler option "CXX" to allow parallel optimization option (e.g. `-O3`), but of which strongly affect the execution speed of the verifications with the use of the SSE/SSE2 instructions in kv itself. Additionally remark that the option of `USE_OPENMP` is available to perform faster interval arithmetic works using an Intel CPU and a SSE/SSE2 operating system.

When you change your computational environment or compile options, it is recommended to **check whether the resulting mode is correctly changed for the kv arithmetic and square root operations**, by compiling and performing testive verifications.

4. List of contents

5. Interval arithmetic (Interval)

6. Double-double precision arithmetics (dd)

7. Interval Arithmetic using numbers / Cheatsheet (dd)(in Japanese)

8. Wrapper for MPFR (mpfr)(in Japanese)

9. Complex number Arithmetic (complex)(in Japanese)

10. Automatic differentiation (autodiff)(in Japanese)

11. Affine Arithmetic (affine)(in Japanese)

12. Power series arithmetic (psa)(in Japanese)

13. Verification by the Kravchuk method for nonlinear equations (in Japanese)

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

15. Verification for initial value problems for ODEs (in Japanese)

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

17. Numerical integration (in Japanese)

18. Verification for special functions (in Japanese)

19. Other functions (in Japanese)

20. Function objects and arithmetic description (in Japanese)

21. Changeable Roundoff Mode and Compiler Options

22. Appendix

23. What is kv?(in Japanese)

24. What's new?(in Japanese)

25. How to use kv?(in Japanese)

26. How to install kv?(in Japanese)

27. Web Demonstration of kv Library

28. Application Examples of kv Library (in Japanese)

29. Papers or Articles related to kv Library (in Japanese)

30. Concluding remarks

This library is based on the C++ code of [libinterval](#) and contributes greatly to its high speed. It is also used for research by students in my laboratory and development is sustained and ongoing there regularly.

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, the Foundation Funding of FY2016 Research Promotion Foundation got a great deal of cooperation. I express my gratitude here.

Copyright (c) 2013-2024 Masahito Kashiwagi. This software is licensed under the [MIT license](#).

[Update history \(in Japanese\)](#)

[Acknowledgment \(in Japanese\)](#)

kv is a C++ Library for Verified Numerical Computation. [kv@kashiwagi.jp](#)

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

Why C++?

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

```
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.

Numerical Linear Algebra in kv Library

- In numerical verification of linear computation, for example, by computing matrix multiplication $C = A \times B$ like
 - ① change rounding mode to downward and compute $\underline{C} = A \times B$
 - ② change rounding mode to upward and compute $\overline{C} = A \times B$we can extremely reduce the number of changing rounding mode and we can use fast ready-made BLAS libraries for matrix multiplication.
- Since the kv library emphasized the natural availability of numeric types other than double, **the current version does not use such kind of techniques at all.**

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

$$\text{Calculate } s = \sum_{k=1}^{1000} \frac{1}{k} .$$

```
#include <kv/interval.hpp> // interval arithmetic
#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-0.4.60.tar.gz
$ tar xzf kv-0.4.60.tar.gz
$ ls
kv-0.4.60/ kv-0.4.60.tar.gz
$ cd kv-0.4.60
$ 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/
$ g++ -I. -O3 interval.cc
$ ./a.out
[7.485470860549956,7.4854708605508238]
```

sample program of double-double interval arithmetic

Calculate $s = \sum_{k=1}^{1000} \frac{1}{k}$ via double-double.

```
#include <kv/interval.hpp> // interval arithmetic
#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";
}
```

[7.485470860550344912656518204308257, 7.485470860550344912656518204360964]

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. : $1234 + 5.432 \rightarrow 1239 + 0.432$)
- **twoproduct**: an algorithm to convert **product** of two floating point numbers to the sum of two non-overlapping floating point numbers.
(e.g. : $1234 \times 5.432 \rightarrow 6703 + 0.088$)
- By using twosum and twoproduct, it is possible to realize a **pseudo quadruple precision arithmetic** 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**.

- a simple wrapper of the MPFR library to perform high precision floating point arithmetic.
- `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 library, **only addition, subtraction, multiplication, division and square root of MPFR are used**, and **excellent mathematical functions of MPFR are not used**.

Policy of implementing interval arithmetic

- In IEEE 754 standard, **only addition, subtraction, multiplication, division 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.

Mechanism for switching internal type of interval arithmetic (1)

- Currently, the kv library supports three types of interval arithmetic: `interval<double>`, `interval<dd>` and `interval<mpfr<N>>`.
- For each type, `double`, `dd` and `mpfr<N>`, there are different ways to perform directed rounding. How to implement directed rounding for unknown types?

Define method of directed rounding according to the interval type

- `template class rop<T>` (rounding operations) is provided, which has 10 member functions for addition, subtraction, multiplication, division and square root with upward and downward rounding (`add_up`, `mul_down`, etc.). The functions defined here do not use directed rounding and cannot be used for verified numerical computation as it is.
- For the types `double`, `dd`, `mpfr<N>`, which we want to use as internal types for interval arithmetic, we specialize the template class by creating `class rop<double>`, `class rop<dd>`, `class rop<mpfr<N>>`, and describe the 10 different arithmetic methods for each type in detail. `rdouble.hpp`, `rdd.hpp`, `rmpfr.hpp` are examples of this.
- Mathematical functions are implemented using only these ten types of operations. Therefore, if you use a high-precision type for internal type of interval, you will automatically get high-precision (according to the type) mathematical functions.
- Types other than `double`, `dd`, `mpfr<N>` can also be used as internal type of interval arithmetic by creating a file that specializes the template class similarly.

Mechanism for switching internal type of interval arithmetic (2)

Excerpt from interval.hpp

```
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;
    }
};
```

Mechanism for switching internal type of interval arithmetic (3)

Excerpt from rdouble.hpp

```
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

Prove existence of solution to $\begin{cases} x^2 - y - 1 = 0 \\ (x - 2)^2 - y - 1 = 0 \end{cases}$ near $(1.01, 0.01)$.

```
#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]([0.9999999999999977,1.0000000000000005],[-3.1225022567584106e-17,1.9081958235745036e-16])
```

Finding all solutions of nonlinear equations

Find all solutions of $\begin{cases} xy - \cos y = 0 \\ x - y + 1 = 0 \end{cases}$ in $x, y \in [-1000, 1000]$.

```
#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]([[[0.2511518352207645, 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 arithmetic can suppress overestimation of interval arithmetic. Instead it takes more computation time.
- Since dependency information with respect to other variables is held, overestimation can be suppressed.
- All variables are expressed in **affine form** like

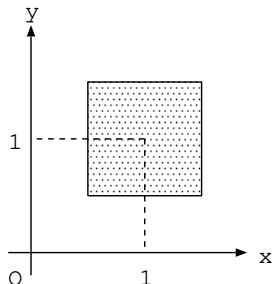
$$x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n .$$

ε_i are **dummy variables** which satisfy $-1 \leq \varepsilon_i \leq 1$ and the coefficients x_i have dependency information.

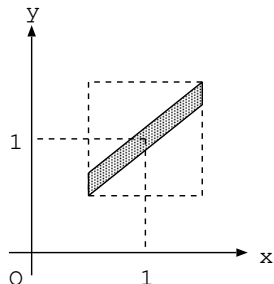
- Every time a nonlinear operation such as multiplication, division or mathematical function appears, **the number of dummy variables increases** and the calculation speed slows down.
- Like as the `interval`, we can use `dd` or `mpfr` as well as `double` for internal type of `affine`.
- has a function to reduce the number of dummy variables.

Dummy variables ε express the dependency information

$$\begin{aligned}x &= 1 + 0.5\varepsilon_1 \\y &= 1 + 0.5\varepsilon_2\end{aligned}$$



$$\begin{aligned}x &= 1 + 0.5\varepsilon_1 \\y &= 1 + 0.4\varepsilon_1 + 0.1\varepsilon_2\end{aligned}$$



The ranges of x, y are the same, but their "joint range" is different.

Conversion between interval

interval \rightarrow affine

$$\begin{pmatrix} [\underline{x}_1, \overline{x}_1] \\ [\underline{x}_2, \overline{x}_2] \\ \vdots \\ [\underline{x}_n, \overline{x}_n] \end{pmatrix} \Rightarrow \begin{pmatrix} \frac{\overline{x}_1 + \underline{x}_1}{2} + \frac{\overline{x}_1 - \underline{x}_1}{2} \varepsilon_1 \\ \frac{\overline{x}_2 + \underline{x}_2}{2} + \frac{\overline{x}_2 - \underline{x}_2}{2} \varepsilon_2 \\ \vdots \\ \frac{\overline{x}_n + \underline{x}_n}{2} + \frac{\overline{x}_n - \underline{x}_n}{2} \varepsilon_n \end{pmatrix}$$

affine \rightarrow interval

$$x = a_0 + a_1 \varepsilon_1 + \cdots + a_n \varepsilon_n$$

$$\Rightarrow [a_0 - r, a_0 + r], \quad \left(r = \sum_{i=1}^n |a_i| \right)$$

Linear Operation is easy

$$x = x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n$$

$$y = y_0 + y_1\varepsilon_1 + \cdots + y_n\varepsilon_n$$

Addition, Subtraction

$$x \pm y = (x_0 \pm y_0) + (x_1 \pm y_1)\varepsilon_1 + \cdots + (x_n \pm y_n)\varepsilon_n$$

$$x \pm \alpha = (x_0 \pm \alpha) + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n \quad .$$

Constant Multiplication

$$\alpha x = (\alpha x_0) + (\alpha x_1)\varepsilon_1 + \cdots + (\alpha x_n)\varepsilon_n \quad .$$

Nonlinear Unary Operation (Mathematical Function)

f : unary operator such as \exp, \log, \dots .

Consider Calculating $z = f(x)$ for an affine variable x .

$$x = x_0 + x_1\varepsilon_1 + \dots + x_n\varepsilon_n$$

- 1 Calculate I (range of x) as follows:

$$I = [x_0 - r, x_0 + r], \quad r = \sum_{i=1}^n |x_i| \quad ,$$

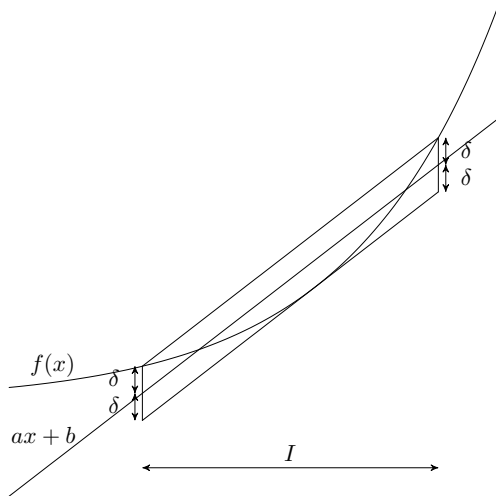
- 2 Calculate $ax + b$ (linear approximation of f over I) and maximum error δ as follows:

$$\delta = \max_{x \in I} |f(x) - (ax + b)|$$

- 3 Calculate the result z as follows:

$$z = a(x_0 + x_1\varepsilon_1 + \dots + x_n\varepsilon_n) + b + \delta\varepsilon_{n+1}$$

Linear Approximation $ax + b$ and Error δ



Nonlinear Binary Operation

For binary operator $g(x, y)$, consider linear approximation $ax + by + c$.
(Almost same as in the case of unary operator.)

Multiplication

$$\begin{aligned} z &= y_0(x - x_0) + x_0(y - y_0) + x_0y_0 + r_x r_y \varepsilon_{n+1} \\ &= x_0y_0 + \sum_{i=1}^n (y_0x_i + x_0y_i)\varepsilon_i + \left(\sum_{i=1}^n |x_i| \right) \left(\sum_{i=1}^n |y_i| \right) \varepsilon_{n+1} \end{aligned}$$

Example of Affine Arithmetic

The QRT (Quispel-Roberts-Thompson) map

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

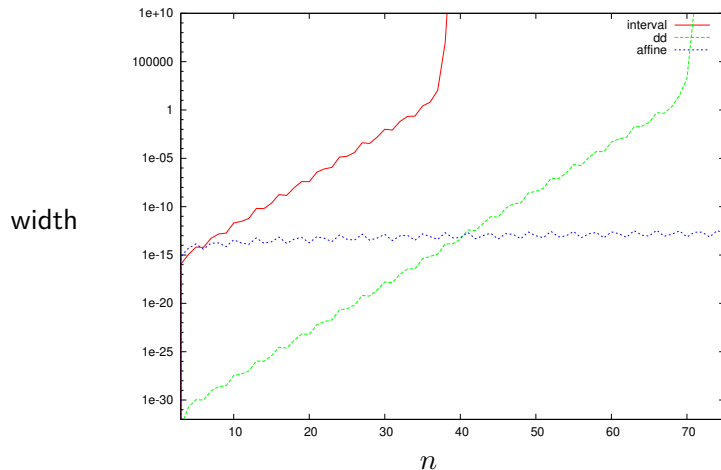
```
#include <kv/interval.hpp>
#include <kv/rdouble.hpp>
int main()
{
    int i;
    kv::interval<double> x, y, z;
    std::cout.precision(17);
    x = 1.;
    y = 1.;
    for (i=2; i<=10000; i++) {
        z = (1 + 2 * y) / (x * y * y);
        std::cout << i << " " << z << "\n";
        x = y;
   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

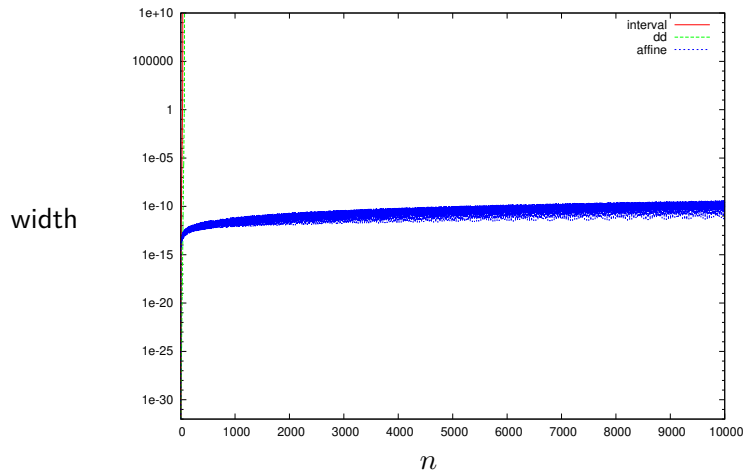
n	interval arithmetic	affine arithmetic
2	[3, 3]	[3, 3]
3	[0.777777777777777767, 0.7777777777777778]	[0.777777777777777756, 0.777777777777777824]
4	[1.408163265306122, 1.4081632653061232]	[1.4081632653061197, 1.4081632653061247]
5	[2.4744801512287302, 2.4744801512287369]	[2.4744801512287271, 2.4744801512287414]
6	[0.68995395922102109, 0.68995395922102732]	[0.6899539592210222, 0.68995395922102621]
7	[2.020393474742363, 2.0203934747424169]	[2.0203934747423817, 2.0203934747423987]
8	[1.7898074714307314, 1.7898074714308816]	[1.7898074714307978, 1.7898074714308153]
⋮	⋮	⋮
31	[0.70098916182277204, 0.70941982097935608]	[0.70519175616865292, 0.70519175616868424]
32	[1.7816188152293368, 1.8444838202503787]	[1.8127715215496742, 1.8127715215497711]
33	[1.890688867011997, 2.1073484458445711]	[1.9960405520559754, 1.9960405520560838]
34	[0.58372124794988644, 0.81879304568504608]	[0.69119381312156691, 0.69119381312160023]
35	[1.5341327531940911, 4.0942614522583929]	[2.4982930525184534, 2.4982930525186054]
36	[0.29640395761996329, 6.6882779916662063]	[1.3900085495715059, 1.390008549571586]
37	[0.0086967943592607538, 106.66548725824453]	[0.78309678534845506, 0.78309678534849637]
38	[1.3369859317919986 $\times 10^{-5}$, 9560542.5436595381]	[3.0105168251706007, 3.0105168251708015]
39	[1.0257053348148149 $\times 10^{-16}$, 1.229984619387229 $\times 10^{19}$]	[0.98924416180811902, 0.98924416180817921]
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, 0.77262518048269758]
43	–	[1.4265918581977079, 1.4265918581978075]
⋮	⋮	⋮
9999	–	[0.76071510659932817, 0.76071510667899534]
10000	–	[1.4727965248961243, 1.4727965251850226]

Results



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

Results ($n \leq 10000$)



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 (multiplication)

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

$$\begin{aligned}(1 + 2t - 3t^2)(1 - t + t^2) &= 1 + t - 4t^2 + 5t^3 - 3t^4 \\ &= 1 + t + (-4 + 5t - 3t^2)t^2 \in 1 + t + [-4, -3.5]t^2\end{aligned}$$

Power Series

$$x_0 + x_1t + x_2t^2 + \cdots + x_nt^n$$

- four basic operations $+$, $-$, \times , \div between power series.
- mathematical functions (\exp , \log , etc and f) 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.

Operation Rules of Type-I PSA (1/4)

$$x(t) = x_0 + x_1t + x_2t^2 + \cdots + x_nt^n$$

$$y(t) = y_0 + y_1t + y_2t^2 + \cdots + y_nt^n$$

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-I PSA (2/4)

multiplication

$$x(t) \times y(t) = z_0 + z_1 t + \cdots + z_n t^n$$

$$z_k = \sum_{i=0}^n x_i y_{k-i}$$

(stop at the n th-order term and do not calculate terms of order $n + 1$ and beyond.)

example of multiplication

Truncate

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

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

$$x(t) \times y(t) = 1 + t - 4t^2 + 5t^3 - 3t^4$$

up to the second-order term, and let

$$x(t) \times y(t) = 1 + t - 4t^2$$

be the result.

Operation Rules of Type-I PSA (3/4)

Mathematical functions such as \sin , etc

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

$$\begin{aligned} &g(x_0 + x_1t + \cdots + x_nt^n) \\ &= g(x_0) + \sum_{i=1}^n \frac{1}{i!} g^{(i)}(x_0)(x_1t + \cdots + x_nt^n)^i \end{aligned}$$

All additions and multiplications in above calculation are executed by the **Type-I PSA**.

Operation Rules of Type-I PSA (4/4)

division

$$x \div y = x \times (1/y) \text{ (reciprocal function and multiplication)}$$

indefinite integral

$$\int_0^t x(t)dt = x_0t + \frac{x_1}{2}t^2 + \dots + \frac{x_n}{n+1}t^{n+1}$$

Power Series

$$x_0 + x_1t + x_2t^2 + \cdots + x_nt^n$$

- 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)

$$x(t) = x_0 + x_1t + x_2t^2 + \cdots + x_nt^n$$
$$y(t) = y_0 + y_1t + y_2t^2 + \cdots + y_nt^n$$

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

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

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

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

(addition and subtraction are exactly the same as in Type-I PSA.)

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

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

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

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

example of multiplication

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

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

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

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

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 with a **remainder term**:

$$\begin{aligned} &g(x_0 + x_1t + \cdots + x_nt^n) \\ &= g(x_0) + \sum_{i=1}^{n-1} \frac{1}{i!} g^{(i)}(x_0)(x_1t + \cdots + x_nt^n)^i \\ &+ \frac{1}{n!} g^{(n)} \left(\text{hull} \left(x_0, \left\{ \sum_{i=0}^n x_it^i \mid t \in D \right\} \right) \right) (x_1t + \cdots + x_nt^n)^n \end{aligned}$$

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

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

division

$$x \div y = x \times (1/y) \text{ (reciprocal function and multiplication)}$$

indefinite integral

$$\int_0^t x(t)dt = x_0t + \frac{x_1}{2}t^2 + \dots + \frac{x_n}{n+1}t^{n+1}$$

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);

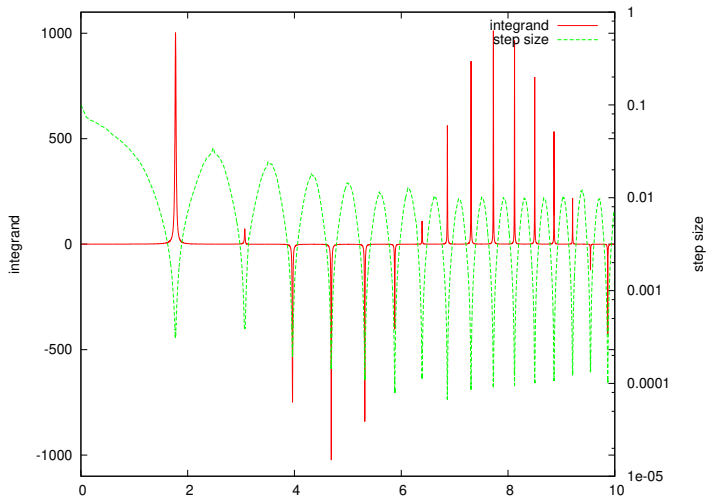
    std::cout << kv::defint_autostep(Func(), (itv)0., (itv)10., 10) << "\n";
}
```

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, 38.38352626464969]
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 oura	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 $t_0 < t_1 < t_2 < \dots$,

- **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 suppressing inflation of interval width.

Overview of 1-step Method (1/3)

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

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

(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)

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

(generating candidate set)

$$1 + \int_0^t (-X_2^2) dt$$
$$= 1 + \int_0^t -(1 - t + t^2)^2 dt$$
$$= 1 + \int_0^t -(1 - 2t + [2.8, 3]t^2) dt$$
$$= 1 - t + t^2 + [-1, -0.933]t^3$$

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)

$$1 + \int_0^t (-Y_c^2) dt$$
$$= 1 - t + t^2 + [-1.14, -0.786]t^3$$

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 $t = t_e$).

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

Variational Equation with respect to Initial Value

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

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

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

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

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

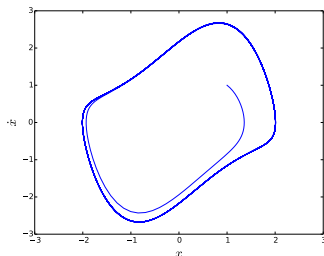
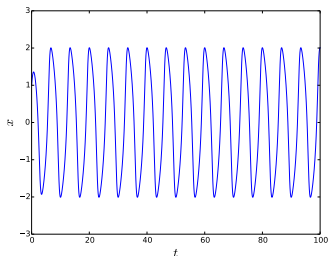
dense output with stepsize 2^{-4} .

```
#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;
    }
};

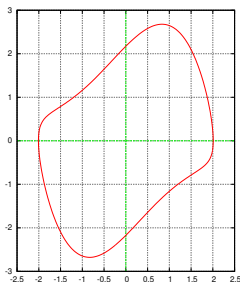
int main()
{
    ub::vector<itv> x;
    itv end;
    std::cout.precision(17);
    x.resize(2);
    x(0) = 1.;
    x(1) = 1;
    end = 100.;
    kv::odelong_maffine(VDP(), x, itv(0.), end, kv::ode_param<double>(), kv::
        ode_callback_dense_print<double>(itv(0.), itv(pow(2., -4))));
}
```

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],[0.8534995323034189,0.85349953230341902])
t: [0.1875,0.1875]
[2]([1.1669406889500346,1.1669406889500352],[0.76674349796008578,0.7667434979600859])
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],[0.07045240241356479,0.070452402414533405])
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 solution of van der Pol Equation ($\mu = 1$) using Krawczyk Method for Poincaré Map (Poincaré section: $x = 0$). The Verified period is

$$T \in [6.6632868593231044, 6.6632868593231534]$$

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]]
```

Finding all solutions of nonlinear equations (1/2)

2-transistor circuit equation with five solutions

$$\begin{pmatrix} 1 & -0.5 & 0 & 0 \\ -0.99 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.5 \\ 0 & 0 & -0.99 & 1 \end{pmatrix} \begin{pmatrix} 10^{-9}/0.99 \times (\exp(x_1/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_2/0.053) - 1) \\ 10^{-9}/0.99 \times (\exp(x_3/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_4/0.053) - 1) \end{pmatrix} \\ + 10^{-4} \begin{pmatrix} 4 & -3 & -2 & 1 \\ -3 & 3 & 1 & 0 \\ -2 & 1 & 4 & -3 \\ 1 & 0 & -3 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} -0.001 \\ 0.000936 \\ -0.001 \\ 0.000936 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

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

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

Finding all solutions of nonlinear equations (2/2)

- A conjecture "2-transistor circuits have at most three solutions." was known, but this paper gave an example with five solutions that negated the conjecture.
- Using all solution program (`allsol.hpp`) in kv, the program executed 295688 non-existence test and 145259 existence test, and succeeded in proving the existence of five solutions.

verified solutions

	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]

Compile options of kv library

- (without compile options) use `fesetround` to change rounding mode. It works in wide environments.
- `-DKV_FASTROUND` use SSE2 (`_mm_setcsr`) to change rounding mode. It works only in Intel x64 environment
- `-DKV_NOHWROUND` don't use hardware rounding change and emulate directed rounding by using nearest rounding mode only. It works in very wide environment, but slow.
- `-DKV_USE_AVX512` use AVX-512 to change rounding mode. Of course, it works only in the latest Intel CPU with AVX-512.
- `-DKV_USE_TPFMA` use FMA for twoproduct.

Comparison of Calculation Time by Finding All solutions of “five-solution” Problem

Environment

Intel Core i9 7900X 3.3GHz/4.3GHz, Memory 128G

Ubuntu 16.04 LTS

gcc 5.4.0

Calculation Time

Accuracy	Compile Option	Calculation Time	
		without -DKV_USE_TPFGMA	with -DKV_USE_TPFGMA
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	



THE
NOBEL
PRIZE

The Nobel Prize in Physics 2021

Explore

The Nobel Prize in Physics 2021



Scientific Background on the Nobel Prize in Physics 2021

"FOR GROUNDBREAKING CONTRIBUTIONS TO OUR
UNDERSTANDING OF COMPLEX PHYSICAL SYSTEMS"

The Nobel Committee for Physics



Ill. Niklas Elmehed ©
Nobel Prize Outreach
Syukuro Manabe
Prize share: 1/4



Ill. Niklas Elmehed ©
Nobel Prize Outreach
Klaus
Hasselmann
Prize share: 1/4



Ill. Niklas Elmehed ©
Nobel Prize Outreach
Giorgio Parisi
Prize share: 1/2

Scientific Background Document begins with . . .

the topic of Lorenz equation and its initial value sensitivity as the origin of complex systems.

upper and lower boundaries [97]. The model is

$$\begin{aligned}\frac{dX}{dt} &= \sigma(Y - X), \\ \frac{dY}{dt} &= X(Ra - Z) - Y \quad \text{and} \\ \frac{dZ}{dt} &= XY - \beta Z,\end{aligned}$$

where X describes the intensity of convective motion, Y is the temperature difference between ascending and descending flow and Z is the deviation from linearity of the vertical temperature profile. The control parameters are the Prandtl Number, σ , which is a property of the fluid, the Rayleigh Number, Ra , which is the dimensionless buoyancy driving vertical fluid motions, and a constant factor β , characterizing the domain geometry.

The Lorenz system acts as a rich toy model of low-dimensional chaos. Since its origin the breadth and extension of studies has been so broad [e.g., 103] it would be difficult to enumerate them all. Key here are the facts that the solutions are bounded, (Fig. 1) and yet exhibit *sensitive dependence on initial conditions* (Fig. 2).

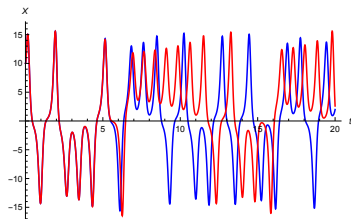
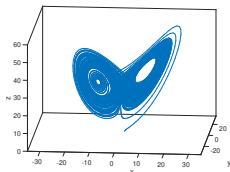
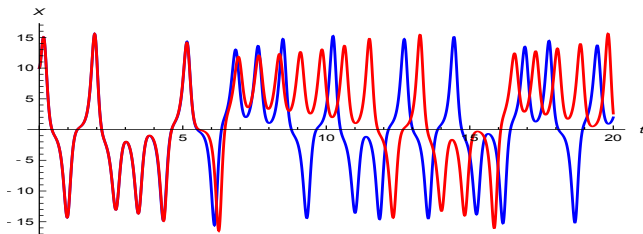


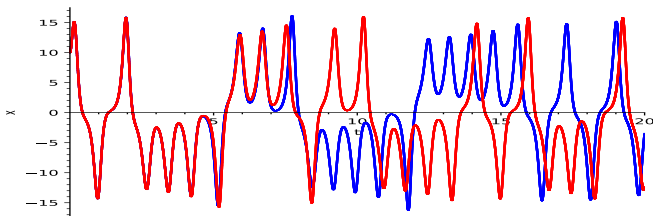
FIG. 2. Plot of $X(t)$ of the Lorenz system with $(\sigma, \beta, Ra) = (10, 8/3, 24.9)$ in which the initial data for all three variables are 10 (blue) or 10.01 (red). The divergence of the two solutions with slightly different initial conditions begins at $t = 5.5$; this is sensitive dependence on initial conditions, often whimsically referred to as the “Butterfly Effect”.

[Butterfly Effect] The blue graph (initial value: 10) and the red graph (initial value: 10.01) show completely different trajectories after a certain time.

Recalculation of the trajectory with guaranteed accuracy



(FIG.2 of "Scientific background on the Nobel Prize in physics 2021")



(Verified Solution of Lorenz Equation calculated by kv library)

The error in numerical calculation is much larger than the error caused by the difference of initial values!

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 restricted 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 arithmetic, Power Series Arithmetic (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 hope you will use of kv library for your research!