精度保証付き数値計算と kv ライブラリ

柏木 雅英
kashi@waseda.jp
http://verifiedby.me/

早稲田大学 基幹理工学部 応用数理学科

2025 年 3 月 15 日

概要

- 数値計算の誤差
- 精度保証付き数値計算について
- 区間演算とその実装
- Krawczyk 法
- kv ライブラリの紹介
  - 概要
  - 区間演算
  - 自動微分、4 倍精度演算、mpfr、affine arithmeric
  - ベキ級数演算 (psa)
  - 数値積分
  - 常微分方程式

数値計算

方程式は解けない

人類は現象を方程式で記述するという方法論を発見し、それを利用して様々な技術を生み出したり未来を予測したりできるようになった。しかし、ほとんどの方程式はきれいな式の形で解くことはできない。

数値計算

解析的に解くことが不可能な数学上の問題を計算機で数値的に解く手法のこと。

- コンピュータの発展とともに様々な分野で用いられ、現代社会に欠かせない技術である。
  - 気象予報、建築物の強度解析、自動車の衝突、化学反応、流体などのシミュレーション。
  - 天体、ロケットなどの軌道計算。
  - コンピュータグラフィックス。
- 世界最初のコンピュータと呼ばれる ENIAC は、ミサイルの弾道の数値計算を目的に開発された。

計算機の実数の表現 (IEEE 754 倍精度浮動小数点数)

実数の表現

- 6.02×10^{23} のような、「1 に近い数 $\times$ 基数のベキ」の形で記憶する。浮動小数点形式という。
- 2 進数で記憶し、64bit=8byte を使う。
- 64bit を以下のように分割し、

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

e を 2 進整数 ($e = \sum_{i=0}^{10} e_i 2^i$)、$m$ を 2 進小数 ($m = \sum_{i=0}^{51} m_i 2^{i-52}$) として

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

で計算される数値と対応する。

- 計算精度 (有効数字) は、$2^{-52} \simeq 2.22 \times 10^{-16} \simeq 10^{-15.65}$ なのでおよそ 16 桁。

IEEE 754 倍精度フォーマットの詳細

IEEE 754 倍精度フォーマットの詳細

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	$(-1)^s \times (0 + m) \times 2^{-1022}$ (非正規化数)
$1 \leq e \leq 2046$	$(-1)^s \times (1 + m) \times 2^{e-1023}$ (正規化数)	
$e = 2047$	$\pm\infty$	NaN (Not a Number)

	-2^{1024}	-2^{-1022}	-2^{-1074}		2^{-1074}	2^{-1022}	2^{1024}	
$-\infty$	正規化数	非正規化数		-0	$+0$	非正規化数	正規化数	∞

丸め誤差

丸め誤差

- 有効数字がおよそ 16 桁なので、例えば $1 \div 3$ を計算すると

$$1 \div 3 = 0.33333333333333148\ldots$$

のように少し誤差が入る。これを丸め誤差という。

- 2 進法を使っているので、$1 \div 10 = 0.1$ でも

$$1 \div 10 = 0.10000000000000005551\ldots$$

のように誤差が入る。

- 1 回だけの計算なら大したことないが、誤差が積み重なると想像以上に大きく誤差が拡大することがある。

2 次方程式の丸め誤差

2 次方程式

2 次方程式 $ax^2 + bx + c = 0$ の解の公式は、

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

大きな誤差が見られる例

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

のときの大きい方の解を計算する。

- (解の公式) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a} = -0.125$
- (解の公式の分子を有理化)
  $\dfrac{2c}{-b - \sqrt{b^2 - 4ac}} = -0.10000000000000002$

連立一次方程式の誤差

連立一次方程式

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

真の解

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

ガウスの消去法で計算した解

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

間違った解の残差を計算すると ...

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

Rump の例題

Rump の例題 (改)

$a = 77617$、$b = 33096$ に対して、以下の値を計算せよ。

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

計算結果

1.17260396480560302734375 (float)
1.1726039400531786949244406059733592 (double)
1.1726039400531786318588349045201801 (double-double)
1.1726039400531786318588349045201838 (mpfr113)
1.1726039400531786318588349045201838 (binary128)
-0.827396059946821368141165095479816291 (mpfr150) ←true value

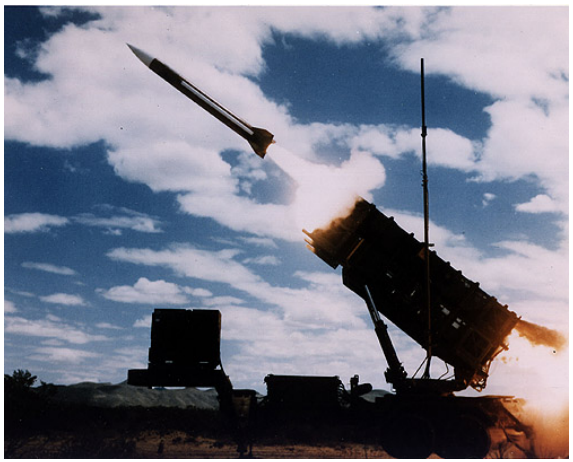
float(仮数部 23bit) から binary128(仮数部 113bit) まで、その精度に応じて計算できているように見えて実はデタラメで、仮数部 150bit でやっと本当の値が見える。

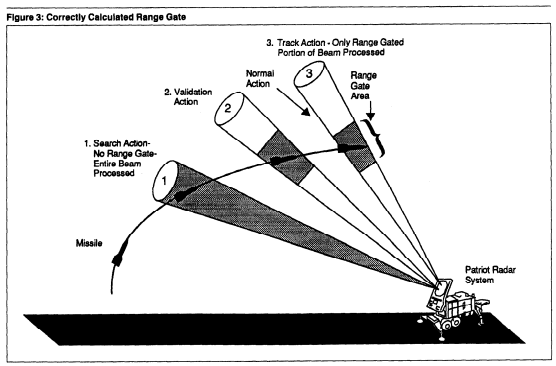
湾岸戦争におけるパトリオットミサイルの事故

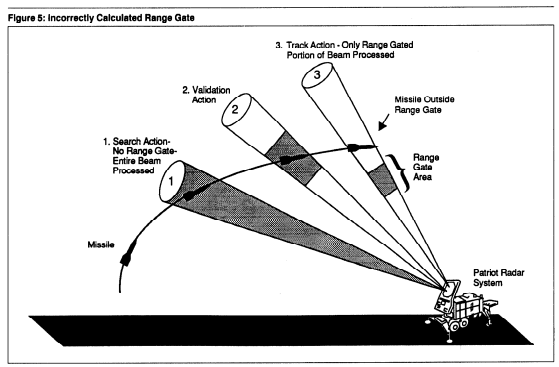
http://www.sydrose.com/case100/298/
https://www.gao.gov/assets/220/215614.pdf

湾岸戦争中、イラク軍のスカッド・ミサイル (Scud Missile) の迎撃のため、アメリカ軍のパトリオット・ミサイル (Patriot Missile) が発射されたが、その内部計算機のわずかな誤差 (0.1 秒を正確に表現できなかったことによる) からスカッド・ミサイルを捉えることが出来ず迎撃は失敗した。

- 発生日時 1991 年 2 月 25 日
- 発生場所 サウジアラビア、Dharan
- 死者 28 名、負傷者約 100 名







精度保証付き数値計算

精度保証付き数値計算

近似解だけでなく、その解の数学的に厳密な誤差評価をも計算する数値計算法。

必要な技術

- 区間演算 (切り捨てと切り上げを併用して計算に混入した誤差を把握する。関数の像の評価も行う。)
- 不動点定理 (不動点定理の成立の十分条件を区間演算で確認することによって、方程式の解の存在と存在範囲を保証する)
- 自動微分も重要。

区間演算 (1/7)

区間演算

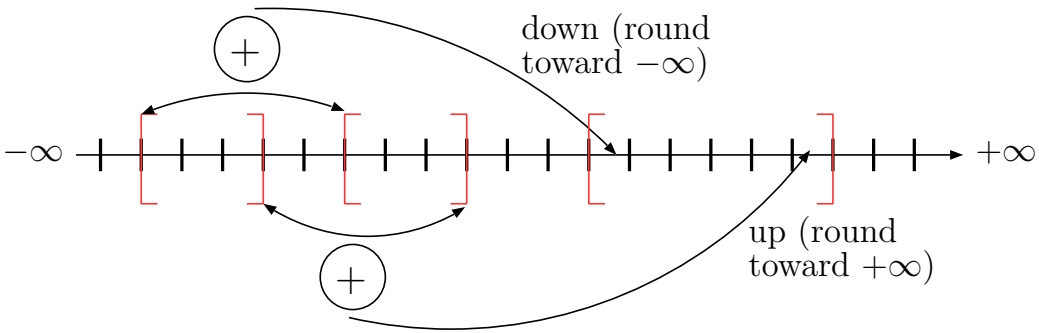
- 精度保証付き数値計算の基本技術。
- 例えば、円周率を $\pi \simeq 3.14$ とするとそれは近似だが、$\pi \in [3.14, 3.15]$ というのは正しい情報。
- 数値を、計算機で表現可能な浮動小数点数を両端に持つ閉区間 $X = [a, b]$ で表現する。
- 区間同士の演算は、集合値演算として計算結果として有り得る値を包含するように行う。
- IEEE754 標準で定義されている「方向付き丸め」を利用する。
- 「丸め誤差の把握」と「関数の値域の評価」の 2 つの役割がある。

区間演算 (2/7)

IEEE754 の丸めモード

- round to nearest: デフォルトの丸めモード。最も近い浮動小数点数に丸める。
- round toward $+\infty$: 上向き丸め
- round toward $-\infty$: 下向き丸め
- round toward 0: 絶対値が大きくならない方向への丸め

区間演算では、下端を下向き丸め、上端を上向き丸めで計算することによって、計算結果が外側に広がるように計算する。



区間演算 (3/7)

- $X = [a, b], Y = [c, d]$
- $\underline{\cdot}, \overline{\cdot}$ はそれぞれ下向き丸め、上向き丸め

- 加算 $X + Y = [a \underline{+} c, b \overline{+} d]$
- 減算 $X - Y = [a \underline{-} d, b \overline{-} c]$
- 乗算 $X \times Y =$

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

- 除算 $X/Y =$

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

($Y \not\ni 0$ の場合のみ定義される)

- 平方根 $\sqrt{X} = [\underline{\sqrt{a}}, \overline{\sqrt{b}}]$

区間演算 (4/7)

10 進数有効数字 3 桁で $(1 \div 3) \times 3$ を計算すると $\cdots$

- 普通の数値計算

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

四捨五入による誤差がでる。

- 区間演算

最初は幅のない区間からスタート。

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

常に区間内に真の値を含みながら計算が進む。

2 次方程式の例を区間演算で計算する

2 次方程式

2 次方程式 $ax^2 + bx + c = 0$ の解の公式は、

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

大きな誤差が見られる例

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

のときの大きい方の解を計算する。

- (解の公式) $\dfrac{-b + \sqrt{b^2 - 4ac}}{2a} = [-0.1875, -0.0625]$
- (解の公式の分子を有理化) $\dfrac{2c}{-b - \sqrt{b^2 - 4ac}} =$
  $[-0.10000000000000004, -0.09999999999999991]$

区間演算 (5/7)

丸めの向きの変更方法

- IEEE754 で丸めの向きを変更可能にすることが要請されているが、その方法は CPU やコンパイラによって違う。
- X86 だと FPU と SSE2 の 2 種類の演算器で丸めの向きの変更方法が異なるなど複雑。
- 最近は C99 準拠のコンパイラが増えたので、`fenv.h` と `fesetround` を使えばハードウェアの違いを吸収してくれる。

```
#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
```

区間演算 (6/7)

区間演算の実装上の注意点

- (入力誤差) プログラム中の定数 (`double x = 0.1;` の「`0.1`」など) はコンパイル時に 10 進数 →2 進数変換が行われ、その丸めの向きは制御できない。
- (出力誤差) 同様に、`std::cout << x << std::endl;` などの表示のときの 2 進数 →10 進数変換の丸めの向きも制御できない。
- (コンパイラの最適化) 最適化によって演算の順序が変えられたり定数をコンパイル時に計算されてしまうなどなどして、-O3 などで最適化を強くかけると丸めの向きの変更が効かないことがある。$\Longrightarrow$ `volatile` を使うなどして適切に最適化を抑制する。
- (数学関数) 数学関数で精度が保証されており丸めの向きの変更も出来るのは `sqrt` のみであり、`sqrt` 以外の関数は全く信用できない。

区間演算 (7/7)

区間演算の過大評価

区間演算は、確かに真の値を含むものの、想像以上に区間幅が広がることがある。

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

区間演算の結果は $[-1.39, -0.59]$ となるが、真の像は $[-1, -0.99]$。

過大評価を緩和する手段

平均値形式 $f(I)$ を直接評価するより、$c = \mathrm{mid}(I)$ として

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

を計算した方がよい場合が多い。

affine arithmetic (後述)

Krawczyk 法 (Krawczyk(1969), Kahan(1968))

区間演算を用いて、非線形方程式 $f(x) = 0, \quad f : \mathbb{R}^n \to \mathbb{R}^n$ の解の存在を保証する。

Krawczyk 法

$I \subset \mathbb{R}^n$ は区間ベクトル (候補者集合)、$c = \mathrm{mid}(I)$、$R \simeq f'(c)^{-1}$、E は単位行列とし、

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

としたとき、$K(I) \subset \mathrm{int}(I)$ ならば I に $f(x) = 0$ の解が唯一存在する。
($\mathrm{int}(I)$: I の内部)

証明

$g(x) = x - Rf(x)$ に対して平均値形式と縮小写像原理を適用する。

Krawczyk 法は $f'(I)$ (区間 I における f の全てのヤコビ行列を包含する区間行列) を持つ $\Longrightarrow$ 自動微分が必要

Krawczyk 法の例

例題

方程式 $\begin{cases} x_1^2 + x_2^2 - 1 = 0 \\ x_1 - x_2 = 0 \end{cases}$ の解が区間 $I = \begin{pmatrix} [0.6, 0.8] \\ [0.6, 0.8] \end{pmatrix}$ にあることを示す。

計算例

$c = \begin{pmatrix} 0.7 \\ 0.7 \end{pmatrix}$、$R = \begin{pmatrix} 0.4 & 0.5 \\ 0.4 & -0.5 \end{pmatrix} \simeq f'(c)^{-1}$ とすると、

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

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

なので、解の一意存在が保証された。

機械的にできるのが嬉しい!

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 ライブラリ

- http://verifiedby.me/kv/ で公開中。
- 作成開始は 2007 年秋頃。公開開始は 2013 年 9 月 18 日。最新版は version 0.4.58。
- 言語は C++。boost C++ Libraries (http://www.boost.org/) も必要。
- 全てヘッダファイルで記述されており、インストールはヘッダファイルをどこかに配置するだけ。
- オープンソースである。精度保証付き数値計算の結果が「証明」であると主張するならば、計算に使われたプログラムは必ず公開されているべき。
- 計算に使う数値の型は double に制限されていない。C++のテンプレート機能を用いて容易に変更することが出来る。

kv ライブラリで扱える数値型

kv ライブラリで扱える数値型

- double
- 区間演算 (多数の精度保証された数学関数含む)
- 4 倍精度 (double-double) 演算
- MPFR ラッパー
- 複素数演算
- 自動微分
- affine arithmeric
- ベキ級数演算
- と、それらの型の組み合わせ

数値型の組み合わせ

例えば、「autodif<interval<dd>>」は、「自動微分型の内部型として区間型を用い、区間型の内部型は double-double」を意味する。

kv ライブラリで実装されたアプリケーション

kv ライブラリで実装されたアプリケーション

- Krawczyk 法による非線形方程式の精度保証
- 非線形方程式の全解探索
- 常微分方程式の初期値問題の精度保証
- 常微分方程式の境界値問題の精度保証
- 精度保証付き数値積分 (1, 2 次元)
- 端点特異性を持つ関数の精度保証付き数値積分
- ガンマ関数、ベッセル関数などの精度保証付き特殊関数
- KKT 方程式を用いた非線形最適化問題の精度保証
- その他

kv ライブラリの web ページ

English | 日本語

最終更新: 2024/8/27

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84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 22.04 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 22.04 (64bit) + gcc 11.2.0
- ubuntu 22.04 (64bit) + clang 14.0.0
- ubuntu 20.04 (64bit) + gcc 9.3.0
- ubuntu 20.04 (64bit) + clang 10.0.0
- 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
- M1 MacBook Air + Mac OS Monterey + gcc 12.2/clang 14.0.6
- Mac OS X Yosemite + gcc
- Mac OS X snow leopard + gcc
- ubuntu 12.04 on MK802(ARM) + gcc
- raspberry pi + raspbian + gcc
- Sharp NetWalker PC-Z1 + ubuntu 9.04 + gcc
- Intel Edison + gcc
- Debian 10 (buster) on chromebook (ARM) + gcc

3. ダウンロードとインストール

ダウンロード: kv-0.4.57.tar.gz (2024年8月27日公開)
(古いversionはこちら)

また、 https://github.com/msksashj/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)

6. 4倍精度演算 (dd)

7. Intel 80bit浮動小数点数 (Float64x, ddx)

8. MPFRラッパー

9. 複素数演算 (complex)

10. 自動微分 (autodif)

11. Affine Arithmetic (affine)

12. ベキ級数演算 (psa)

13. Krawczyk法による非線形方程式の解の精度保証

14. 非線形方程式の全解探索

15. 常微分方程式の初期値問題の精度保証

16. 初期値問題solverと射撃法による境界値問題の精度保証

17. 数値積分

18. 特殊関数の精度保証

19. その他の機能

20. 関数オブジェクトによる問題の記述

21. 丸めモードの変え方とコンパイルオプションまとめ

22. その他

22.1 boostとは

22.2 行列計算 (boost.ublas)

22.3 数値型のプログラミング

22.4 おまけ: シンプルな区間演算ライブラリ

23. 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。

24. 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。

25. kvライブラリに関連した論文、記事など

kvライブラリに言及のある論文、記事などを集めました。

26. 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。

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

http://verifiedby.me/kv/

kv ライブラリの web ページ (英語版)

English | 日本語

Last Update: Aug. 27, 2024

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 (84 pages)

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 22.04 (64bit) + gcc. Unix operating systems (64bit) are recommended, while it was confirmed that kv library works on the following computational environments:

- ubuntu 22.04 (64bit) + gcc 11.2.0
- ubuntu 22.04 (64bit) + clang 14.0.0
- ubuntu 20.04 (64bit) + gcc 9.3.0
- ubuntu 20.04 (64bit) + clang 10.0.0
- 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
- M1 MacBook Air + Mac OS Monterey + gcc 12.2/clang 14.0.6
- 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
- Debian 10 (buster) on chromebook (ARM) + gcc

3. Download and Install

Download: kv-0.4.57.tar.gz (updated on Aug. 27, 2024)
(Old versions are available to download here.)

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

The 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 the kv library, the three directory of "kv", "test" and "example" 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 it 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++ -L. -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 (interval)

6. Double-double precision arithmetics (dd)

7. Intel 80bit floating point numbers (Float64x, ddx) (in Japanese)

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

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

10. Automatic differentiation (autodif) (in Japanese)

11. Affine Arithmetic (affine) (in Japanese)

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

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

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

15. Verification for initial value problem 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 problem description (in Japanese)

21. Changing Rounding Mode and Compiler Options

22. Appendix

22.1 What is Boost? (in Japanese)

22.2 Matrix computation with boost.ublas (in Japanese)

22.3 Numeric types (in Japanese)

22.4 Bonus: a simple interval arithmetic library

23. Web Demonstration of kv Library

24. Application Examples of kv Library (in Japanese)

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

26. 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.

Update history (in Japanese)

Acknowledgment (in Japanese)

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

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

なぜ C++を選んだか?

- 同じ表記のプログラム (例えば次のようなもの) に対して、演算子多重定義を使って

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

  - double
  - interval
  - 自動微分型
  - 内部が interval な自動微分型
  - ベキ級数型
  - 多倍長数
  - 多倍長数 interval
  - etc

  など、様々な特殊な動作をする異なる「数値型」を使いたい。
- python, ruby, matlab などの「型の無い」言語を使って記述すると楽だが、実行時に演算を行う度に内部では型の判定が行われることになり、速度が低下する。
- C++の template 機能を使えば、型を仮定しない generic な記述を行いながら、実行時ではなくコンパイル時に型の判定を全て終わらせるため、速度が低下しない。

C++のテンプレート機能 (テンプレート関数)

テンプレート無し

```
#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";
}
```

テンプレートあり

```
#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++のテンプレート機能 (テンプレートクラス)

テンプレート無し

```
#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();
}
```

テンプレートあり

```
#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();
}
```

行列ベクトル計算

boost.ublas

- 行列ベクトル計算は、boost (`http://www.boost.org/`) に含まれている ublas を用いている。
- ublas は、行列やベクトルの成分の型がテンプレートになっているので、区間行列等が自然に扱える。
- 名前は ublas だが、BLAS の機能を全て持っているという意味で、BLAS 的な高速性を持つわけではない。

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

区間演算 (interval)

- 上端下端型の区間演算を行う。
- exp, log, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, asinh, acosh, atanh, expm1, log1p, abs, pow などの精度保証付きの数学関数を持つ。
- 10 進文字列と double との丸め方向指定付き相互変換を持ち、正しく入出力が出来る。
- 上端と下端に用いる数値型はテンプレートになっており、double 以外の型も使える。例えば double-double 型や MPFR を使える。ただし、上向き下向き双方の丸めに対応した加減乗除、平方根、文字列との相互変換の方法を定義する必要がある。
- サポートする環境は、C99 準拠の fesetround が使えること。
  - X86 CPU の SSE2
  - 最近点丸めのみを用いた方向付き丸めのエミュレーション
  - 最新の Intel CPU の AVX-512
  - FMA 命令

  を使うオプションもある。

区間演算プログラムの例

$$s = \sum_{k=1}^{1000} \frac{1}{k}$$ を計算する。

```
#include <kv/interval.hpp> // 区間演算
#include <kv/rdouble.hpp> // double の方向付き丸めを定義

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";
}
```

```
[7.4854708605049956,7.4854708605508238]
```

使い方

解凍

```
$ ls
kv-0.4.52.tar.gz
$ tar xfz kv-0.4.52.tar.gz
$ ls
kv-0.4.52/  kv-0.4.52.tar.gz
$ cd kv-0.4.52
$ ls
LICENSE.txt  README.txt  example  kv  test
```

インストール

必要なのは kv ディレクトリ以下。適当な場所 (例えば `/usr/local/include/`) に配置する。

compile & run

```
$ ls
interval.cc  kv/
$ c++ -I. -O3 interval.cc
$ ./a.out
[7.485470860549956,7.4854708605508238]
```

区間演算プログラム (double-double)

$$s = \sum_{k=1}^{1000} \frac{1}{k}$$ を double-double で計算する。

```
#include <kv/interval.hpp> // 区間演算
#include <kv/dd.hpp> // double-double
#include <kv/rdd.hpp> // dd の方向付き丸めを定義

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";
}
```

```
[7.4854708605503449126565182043 08257,7.4854708605503449126565182043 60964]
```

double-double (dd)

- twosum: 2 つの数の和を 2 つの仮数部の重なりのない数の和に変換するアルゴリズム。(例: $1234 + 5.432 \to 1239 + 0.432$)
- twoproduct: 2 つの数の積を 2 つの仮数部の重なりのない数の和に変換するアルゴリズム。(例: $1234 \times 5.432 \to 6703 + 0.088$)
- twosum と twoproduct を組み合わせると、2 つの倍精度数を用いた擬似的な 4 倍精度計算を実現できる。一般的な多倍長演算よりかなり高速。
- 単体 (`dd.hpp`) で使った時は近似計算。
- `dd.hpp` と `rdd.hpp`(方向付き丸めでの `dd` 型の演算を定義) を併用し、`interval` 型の内部型として `dd` 型を使うと、端点に `dd` 型を持つ 4 倍精度区間演算が可能。

mpfr

- 高精度浮動小数点計算が行える有名な MPFR ライブラリの簡単な wrapper。
- 単体 (`mpfr.hpp`) で使った時は近似計算。
- `kv::mpfr<106>`のようにパラメータとして仮数部長を指定して使う。
- `mpfr.hpp` と `rmpfr.hpp` を併用し、`interval` 型の内部型として `mpfr` 型を使うと、端点に `mpfr` 型を持つ区間演算が可能。但し、MPFR の機能を使うのは加減乗除と平方根のみであり、せっかく MPFR が持っている優秀な数学関数群は一切用いられない。

区間演算の実装の方針

- IEEE754 標準で信頼できて丸めの向きの変更ができるのは加減乗除と平方根のみ。
- 上向きまたは下向き丸めの加減乗除と平方根のみを用いてアルゴリズムを構成する。
- 精度保証付き数学関数も同様に上向きまたは下向き丸めの加減乗除と平方根のみを用いて実装する。
- 上向きと下向き丸めの加減乗除と平方根が実装されているならば、double でなくとも区間の両端の数として使用できる。dd や mpfr を内部に持つ区間演算ができるのはそのため。

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?

内部型に応じた方向付き丸めの定義

- 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 10 種類のメンバ関数 (`add_up`, `mul_down` など) を持つ `template class rop<T>` (rounding operations) が用意されている。ここで定義される関数は方向付き丸めを使っておらず、このままでは精度保証付き数値計算には使えない。
- 区間演算の内部型として使いたい型 (`double`, `dd`, `mpfr<N>`) について、`class rop<double>`, `class rop<dd>`, `class rop<mpfr<N>>` を作成することによって template class の特殊化を行い、それぞれの型に対する 10 種類の演算の方法を具体的に記述する。`rdouble.hpp`, `rdd.hpp`, `rmpfr.hpp` がその実例。
- 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。
- `double`, `dd`, `mpfr<N>`以外の型も、同様に template class の特殊化を行うファイルを作成すれば区間演算の内部型として使用することができる。

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;
    }
};
```

Krawczyk 法による非線形方程式の解の精度保証

方程式 $\begin{cases} x^2 - y - 1 = 0 \\ (x-2)^2 - y - 1 = 0 \end{cases}$ に対して $(1.01, 0.01)$ の近くに解があることを示す。

```
#include <kv/kraw-approx.hpp>
namespace ub = boost::numeric::ublas;
struct Func { // 解きたい問題を関数オブジェクトの形で記述
    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; // (1.01, 0.01)を初期値としてニュートン法を3回行い、
    kv::krawczyk_approx(Func(), x, ix, 3, 1); // 候補者集合を作り、解の存在をチェック
}
```

```
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])
```

非線形方程式の全解探索の例

方程式 $\begin{cases} xy - \cos y = 0 \\ x - y + 1 = 0 \end{cases}$ のすべての解を $x, y \in [-1000, 1000]$ で探索する。

```
#include <kv/allsol.hpp>
namespace ub = boost::numeric::ublas;
struct Func { // 解きたい問題を関数オブジェクトの形で記述
    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); // 全解探索
}
```

```
[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]([0.2511518352207645,0.2511518352207607],[1.2511518352207642,1.2511518352207654])(ex:improved)
ne_test: 49, ex_test: 3, ne: 23, ex: 2, giveup: 0
```

自動微分 (autodif)

- Bottom-Up 型の自動微分を実装している。一階微分のみ。(一変数関数であれば、ベキ級数型 (psa) で高階微分を行える。)

```
#include <kv/autodif.hpp>
namespace ub = boost::numeric::ublas;
// 関数の定義
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.;
    // 自動微分型の初期化
    va1 = kv::autodif<double>::init(v1);
    // 関数呼び出し
    va2 = func(va1);
    // 自動微分型を分解
    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 とは

- 区間演算の過大評価を抑制できる。その代わり計算時間がかかる。
- 全ての変数について、入力変数またはノイズに関する依存性を保持するため、区間幅の爆発を防げる。
- 全ての数値は

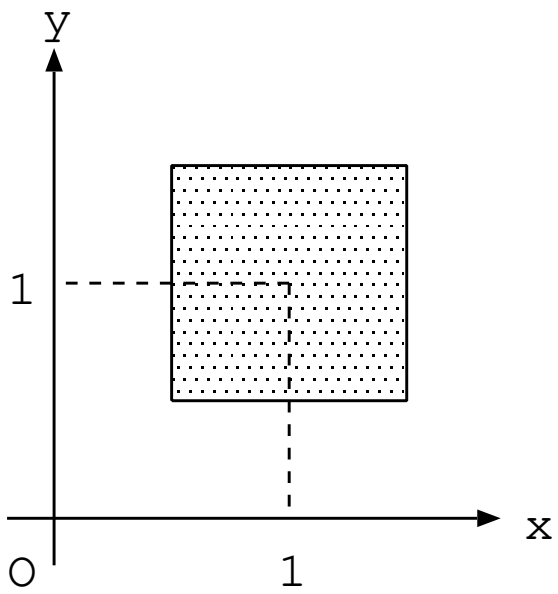
$$x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + \cdots + x_n\varepsilon_n$$

のような Affine 形式で表現される。ε_i は $-1 \leq \varepsilon_i \leq 1$ を動くダミー変数であり、その係数により依存性を表現する。
- 乗除算や数学関数などの非線形演算が出現する度にダミー変数の数が増え、計算が遅くなる。

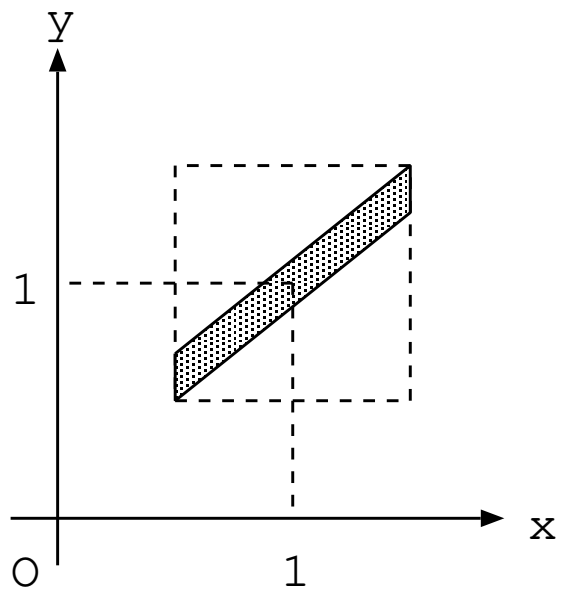
- 内部型は double だけでなく dd や mpfr も入れられる (interval と同様の仕様)。
- ダミー変数の数を削減する機能を持っている。

ダミー変数 ε は異なる変量間の相関性を表現する

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



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



x, y それぞれの変域は同じだが、"joint range" は異なる。

区間との相互変換

区間 → affine

$$\begin{pmatrix} [\underline{x_1}, \overline{x_1}] \\ [\underline{x_2}, \overline{x_2}] \\ \vdots \\ [\underline{x_n}, \overline{x_n}] \end{pmatrix} \Longrightarrow \begin{pmatrix} \frac{\overline{x_1}+\underline{x_1}}{2} + \frac{\overline{x_1}-\underline{x_1}}{2}\varepsilon_1 \\ \frac{\overline{x_2}+\underline{x_2}}{2} + \frac{\overline{x_2}-\underline{x_2}}{2}\varepsilon_2 \\ \vdots \\ \frac{\overline{x_n}+\underline{x_n}}{2} + \frac{\overline{x_n}-\underline{x_n}}{2}\varepsilon_n \end{pmatrix}$$

affine → 区間

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

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

線形計算は簡単

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

加算、減算

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

定数の乗算

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

非線形単項演算 (数学関数)

f: $\exp, \log, \cdots$ などの単項演算
affine 変数 x に対して、$z = f(x)$ を計算することを考える。

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

1. x の変動区間 I を次のように計算する:

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

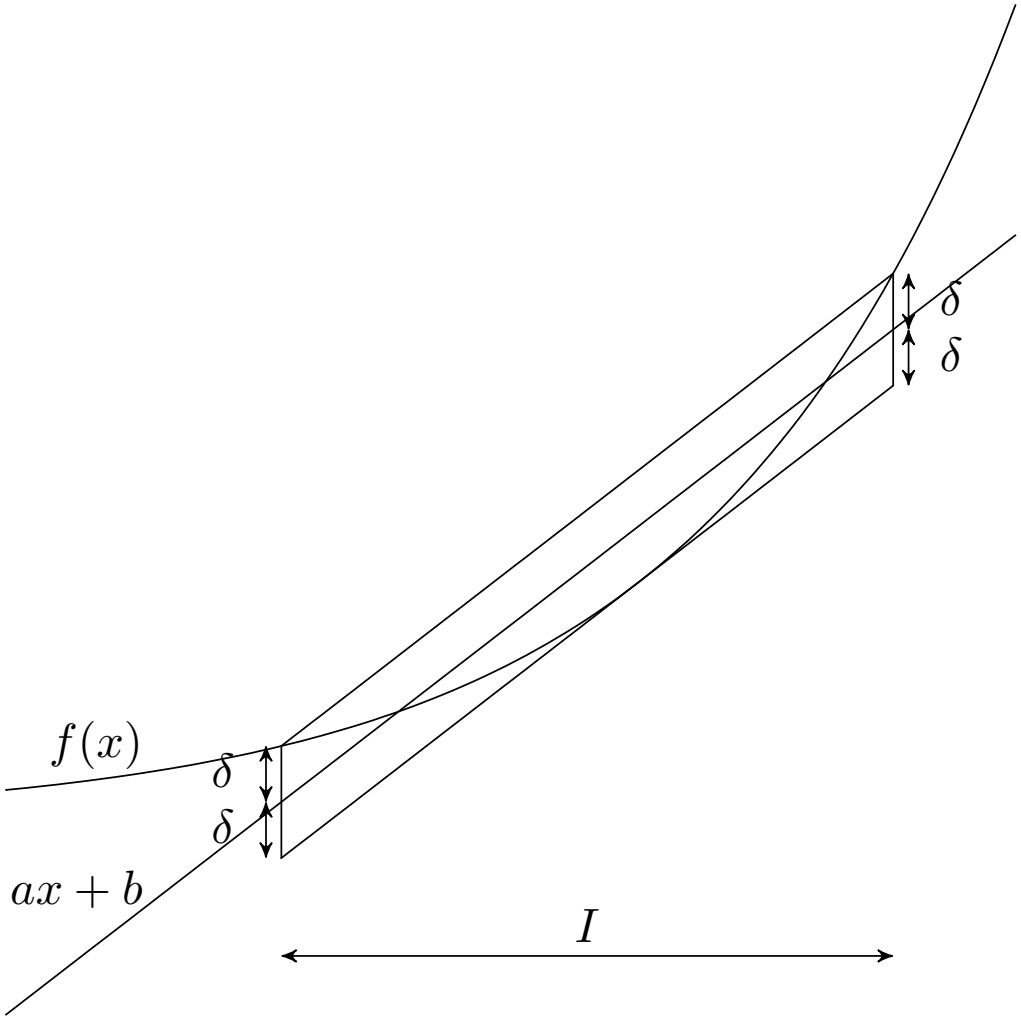
2. f の I 上における線形近似 $ax + b$ と最大誤差 δ を計算する:

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

3. 計算結果 z は次の式で計算する:

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

線形近似 $ax + b$ と誤差 δ



非線形二項演算

二項演算子 $g(x,y)$ に対して、線形近似 $ax+by+c$ を考える。(単項演算の場合とほぼ同様。)

乗算

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

Affine Arithmetic の使用例

QRT(Quispel-Roberts-Thompson) 写像

- 三項間漸化式 $x_{n+1} = \dfrac{1+\alpha x_n}{x_{n-1}x_n^{\sigma}}$
- $\sigma = 2$、$\alpha = 2$、$x_0 = x_1 = 1$ として、区間演算と Affine Arithmeric でどこまで計算できるか試してみた。

```
#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;
    }
}
```

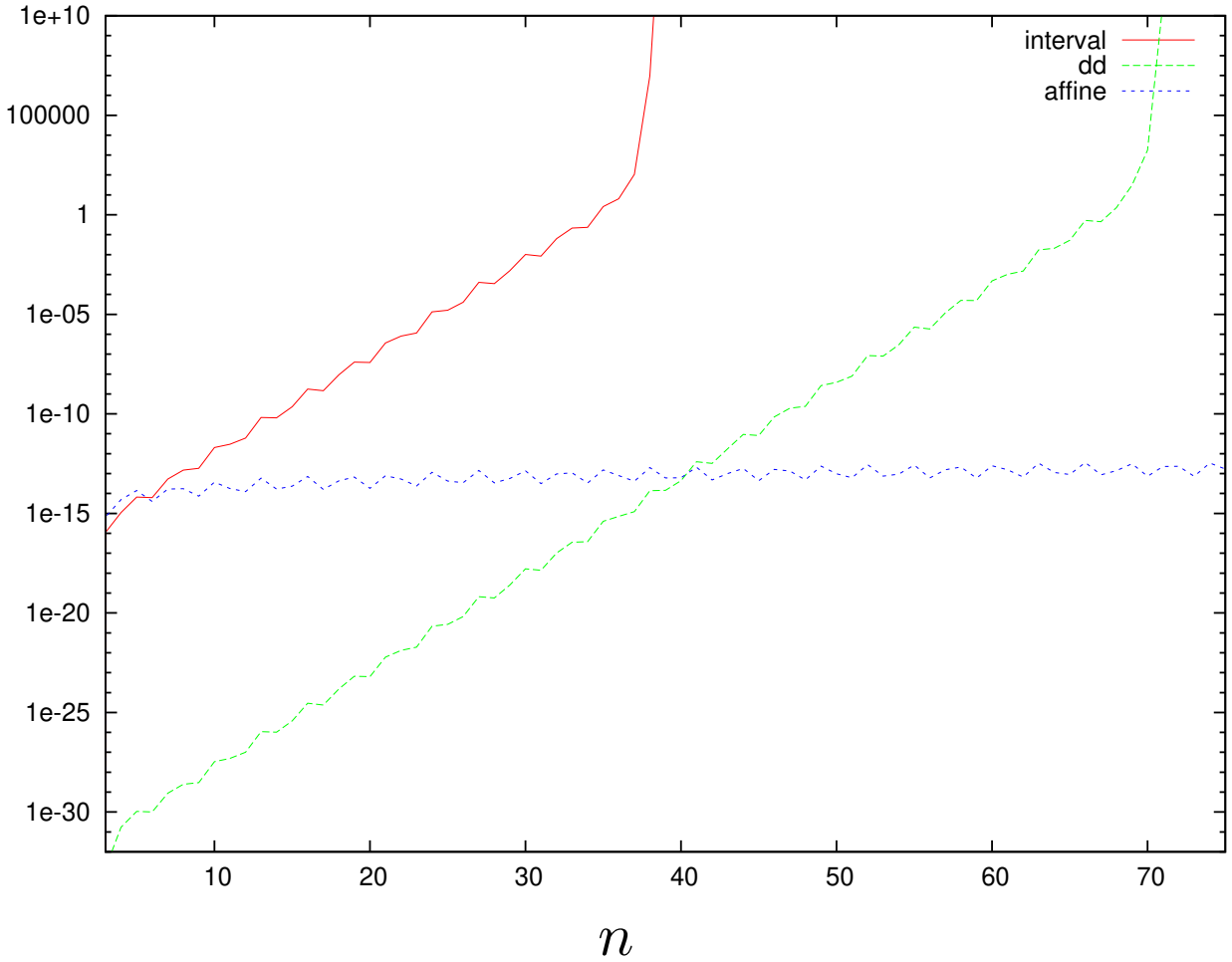
```
#include <kv/affine.hpp>
int main()
{
    int i;
    kv::affine<double> x, y, z;
    std::cout.precision(17);
    x = 1.;
    y = 1.;
    for (i=2; i<=10000; i++) {
        z = (1 + 2 * y) / (x * y * y);
        std::cout << i << " " << to_interval(z)
             << "\n";
        x = y;
       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

区間演算と Affine Arithmeric の計算結果

n	区間演算	Affine Arithmetic
2	$[3, 3]$	$[3, 3]$
3	$[0.77777777777777{\color{red}67}, 0.77777777777777{\color{red}78}]$	$[0.77777777777777{\color{red}56}, 0.77777777777777{\color{red}824}]$
4	$[1.40816326530612{\color{red}2}, 1.40816326530613{\color{red}2}]$	$[1.40816326530611{\color{red}97}, 1.40816326530612{\color{red}47}]$
5	$[2.47448015122873{\color{red}02}, 2.47448015122873{\color{red}69}]$	$[2.47448015122872{\color{red}71}, 2.47448015122874{\color{red}14}]$
6	$[0.68995395922102{\color{red}109}, 0.68995395922102{\color{red}732}]$	$[0.68995395922102{\color{red}2}, 0.68995395922102{\color{red}621}]$
7	$[2.02039347474236{\color{red}3}, 2.02039347474241{\color{red}69}]$	$[2.02039347474238{\color{red}17}, 2.02039347474239{\color{red}87}]$
8	$[1.78980747143073{\color{red}14}, 1.78980747143088{\color{red}16}]$	$[1.78980747143079{\color{red}78}, 1.78980747143081{\color{red}53}]$
⋮	⋮	⋮
31	$[0.70098916182277{\color{red}204}, 0.70941982097935{\color{red}608}]$	$[0.70519175616865{\color{red}292}, 0.70519175616868{\color{red}424}]$
32	$[1.78161881522933{\color{red}68}, 1.84448382025037{\color{red}87}]$	$[1.81277152154967{\color{red}42}, 1.81277152154977{\color{red}11}]$
33	$[1.89068886701199{\color{red}7}, 2.10734844584457{\color{red}11}]$	$[1.99604055205975{\color{red}4}, 1.99604055205608{\color{red}38}]$
34	$[0.58372124794988{\color{red}644}, 0.81879304568504{\color{red}608}]$	$[0.69119381312156{\color{red}691}, 0.69119381312160{\color{red}023}]$
35	$[1.53413275319409{\color{red}11}, 4.09426145225839{\color{red}29}]$	$[2.49829305251845{\color{red}34}, 2.49829305251860{\color{red}54}]$
36	$[0.29640395761996{\color{red}329}, 6.68827799166620{\color{red}63}]$	$[1.39000854957150{\color{red}59}, 1.39000854957158{\color{red}6}]$
37	$[0.00869679435926{\color{red}07538}, 106.665487258244{\color{red}53}]$	$[0.78309678534845{\color{red}506}, 0.78309678534849{\color{red}637}]$
38	$[1.3369859317919986 \times 10^{-5}, 9560542.5436595{\color{red}381}]$	$[3.0105168251706{\color{red}007}, 3.0105168251708{\color{red}015}]$
39	$[1.0257053348148149 \times 10^{-16}, 1.229984619387229 \times 10^{19}]$	$[0.98924416180811{\color{red}902}, 0.98924416180817{\color{red}921}]$
40	$[6.9138205018986439 \times 10^{-46}, 1.7488703313159241 \times 10^{56}]$	$[1.0109923081577{\color{red}889}, 1.0109923081578{\color{red}503}]$
41	$[2.6581843623384974 \times 10^{-132}, 7.1339291414655989 \times 10^{162}]$	$[2.98877382084438{\color{red}05}, 2.98877382084459{\color{red}11}]$
42	$[0, \infty]$	$[0.77262518048264{\color{red}96}, 0.77262518048269{\color{red}758}]$
43	–	$[1.42659185819770{\color{red}79}, 1.42659185819780{\color{red}75}]$
⋮	⋮	⋮
9999	–	$[0.76071510659932{\color{red}817}, 0.76071510667899{\color{red}534}]$
10000	–	$[1.47279652489612{\color{red}43}, 1.47279652518502{\color{red}26}]$

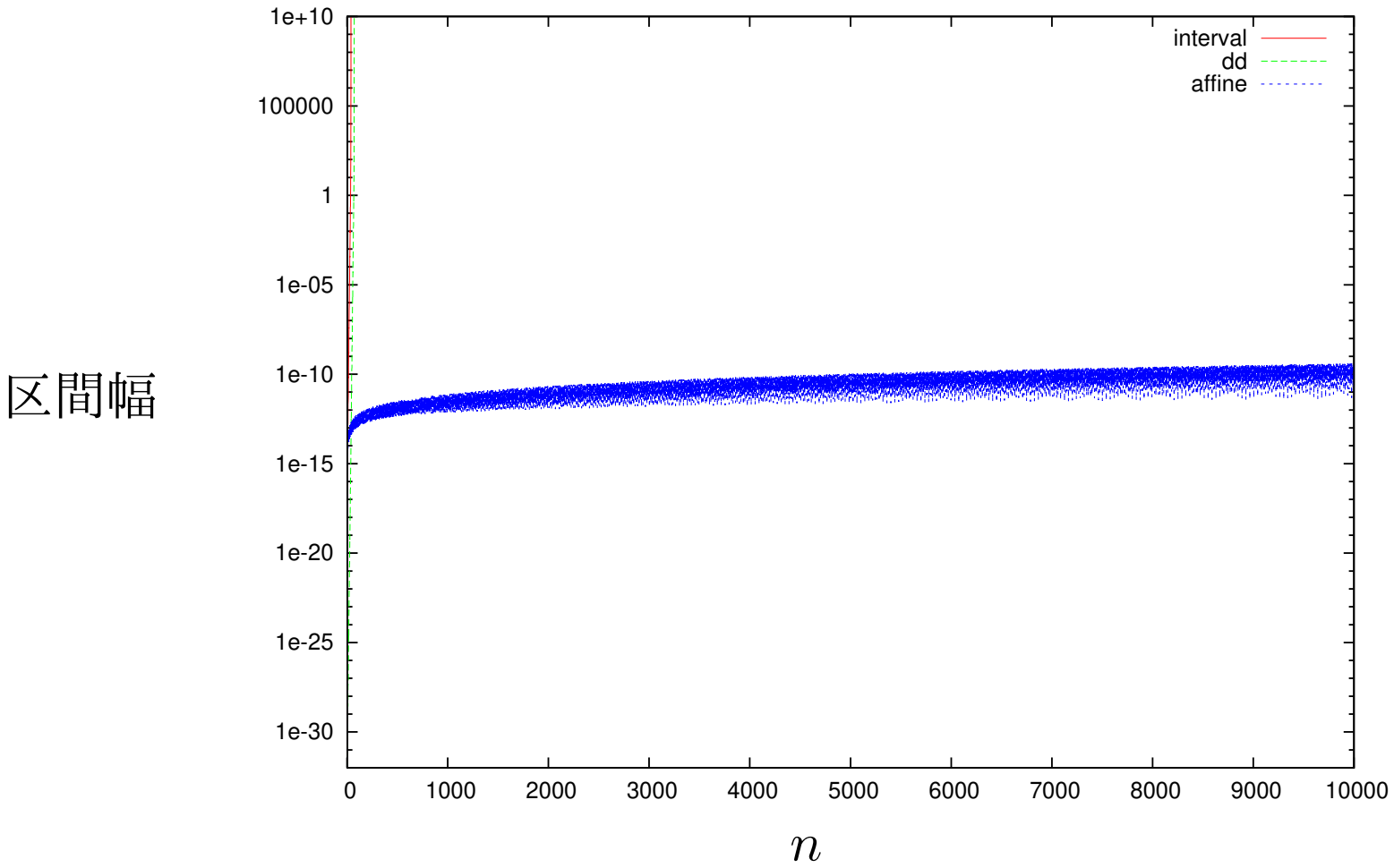
区間幅のグラフ

区間幅



(dd は擬似 4 倍精度区間演算によるもの)

区間幅のグラフ ($n = 10000$ まで)



ベキ級数演算 (psa)

- 常微分方程式の初期値問題や数値積分の精度保証に使う。高階微分の計算にも使える。
- Type-I と Type-II の 2 種類がある。n を整数として、

  Type-I PSA　単に n 次より高次の項を捨てる。

  Type-II PSA　n 次より高次の項の影響を最高次の項 t^n の区間係数に含めさせる。

PSA の例 (積)

$1+2t-3t^2$ と $1-t+t^2$ の積	
Type-I PSA	Type-II PSA
定義域は決めなくてよい	定義域 $=[0,0.1]$
$1+t-4t^2$	$1+t+[-4,-3.5]t^2$

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

Type-I PSA

ベキ級数型

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

- ベキ級数型同士の加減乗除。
- ベキ級数型に対する exp, log, $\int$ などの数学関数。
- 演算結果の n 次までの項を残し、$n+1$ 次以降は切り捨てる。
- 以下とほとんど同じ:
  - Mathematica の ‘`Series`’。
  - Intlab の ‘`taylor`’。
  - 1 変数関数の高階微分を計算する自動微分法。

Type-I PSA の演算規則 (1/4)

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

加減算

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

加算の例

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

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

Type-I PSA の演算規則 (2/4)

乗算

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

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

(n 次までで止め、$n+1$ 次以降の項は計算しない。)

乗算の例

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

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

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

を 2 次の項までで打ち切った、

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

を計算結果とする。

Type-I PSA の演算規則 (3/4)

sin などの数学関数

その関数を g として、

$$\begin{aligned} & g(x_0 + x_1 t + \cdots + x_n t^n) \\ & = g(x_0) + \sum_{i=1}^{n} \frac{1}{i!} g^{(i)}(x_0)(x_1 t + \cdots + x_n t^n)^i \end{aligned}$$

のように g の点 x_0 での Taylor 展開に代入することによって得る。この計算中に現れる加算や乗算は Type-I PSA で行う。

Type-I PSA の演算規則 (4/4)

除算

$x \div y = x \times (1/y)$ と乗算と逆数関数に分解

不定積分

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

Type-II PSA

ベキ級数型

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

- 固定された有限閉区間 $D = [t_1, t_2]$ 上で定義される。
- 演算結果は n 次までしか保持しないが、$n+1$ 次以降の項の影響は n 次の項の係数を区間にすることで吸収する。
- 係数 $x_0, \cdots, x_n$ は区間。
- ただし多くの場合、$x_0, \cdots, x_{n-1}$ は幅の狭い区間、x_n は幅の広い区間。

Type-II PSA の演算規則 (1/5)

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

加減算

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

加算の例

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

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

(加減算は Type-I PSA とまったく同じ)

Type-II PSA の演算規則 (2/5)

乗算

1. まず、打ち切り無しで乗算を行う。

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

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

2. $2n$ 次から n 次に減次する。

m 次から n 次への減次

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

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

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

Type-II PSA の演算規則 (3/5)

乗算の例

定義域を $D = [0, 0.1]$ とする。

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

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

Type-II PSA の演算規則 (4/5)

sin などの数学関数

その関数を g として、

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

のように g の点 x_0 での剰余項付きの Taylor 展開に代入することによって得る。この計算中に現れる加算や乗算は Type-II PSA で行う。

Type-II PSA の演算規則 (5/5)

除算

$x \div y = x \times (1/y)$ と乗算と逆数関数に分解

不定積分

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

精度保証付き数値積分

区間 $[x_i, x_i + \Delta t]$ における積分

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

を次のように計算する。

1. n 次のベキ級数

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

   に対して、

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

   を $[0, \Delta t]$ を定義域とした Type-II PSA で計算する。
2. 計算結果 $y(t)$ を

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

   とすると、積分値は $y(\Delta t)$ で得られる。

ステップ幅 Δt の決定

ε_0 を 1 ステップで混入する誤差の目標値とする。例えば machine epsilon。

1. Type-I PSA を用いて Taylor 展開を計算し、その係数を見て適切なステップ幅 Δt_0 を推定する。Type-I PSA で計算された Taylor 展開を

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

として、

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

とする。
2. ステップ幅 Δt_0 を用いて Type-II PSA を使って精度保証付きで 1 ステップの計算を行う。
3. ステップ幅 Δt_0 で計算して実際に混入した誤差を ε として、新しいステップ幅を

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

で推定する。ただし、n は Taylor 展開の次数。
4. ステップ幅 Δt_1 を用いて Type-II PSA を使って精度保証付きで 1 ステップの計算を行う。

精度保証付き数値積分の例 (1/3)

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

使用プログラム

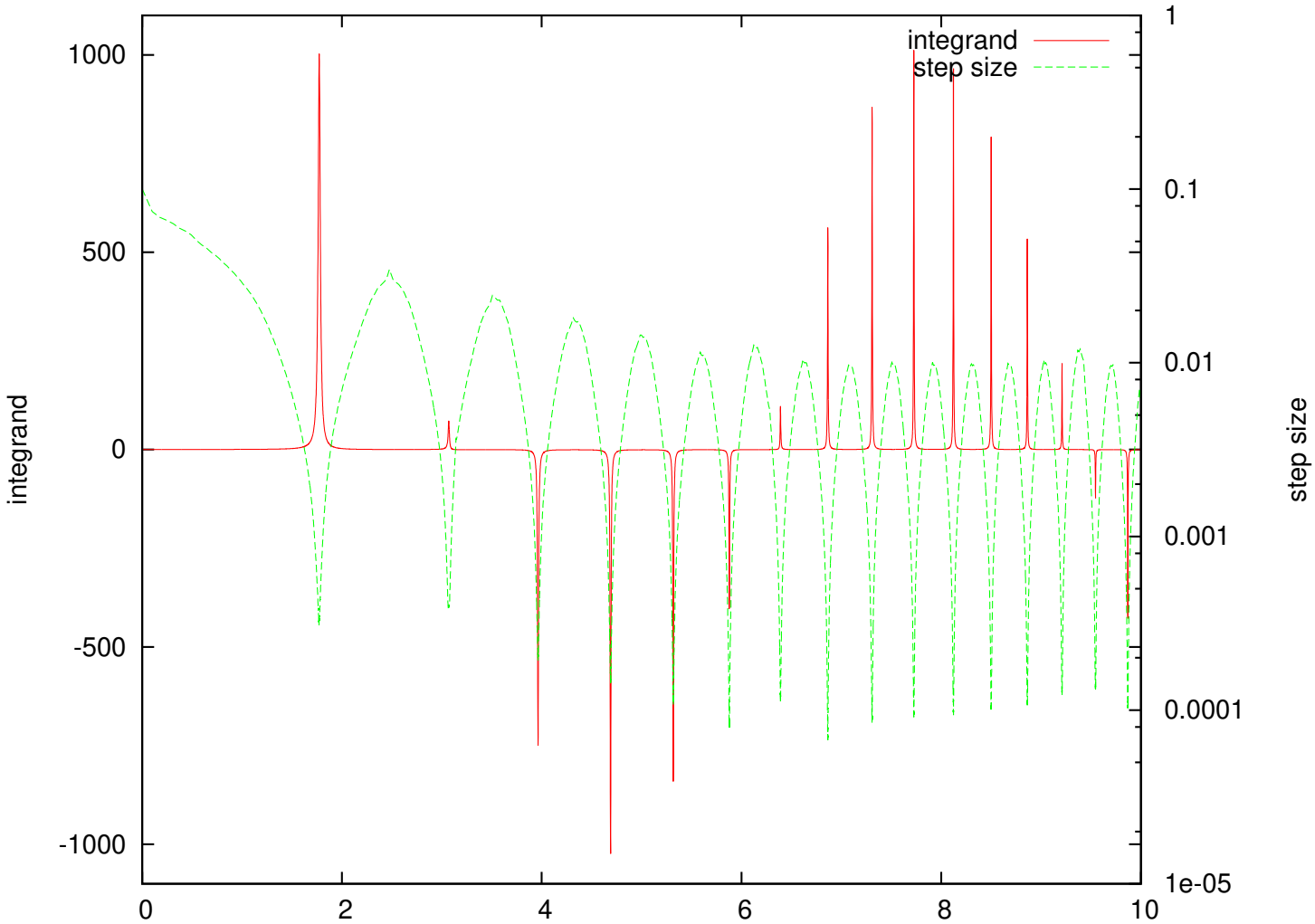
```
#include <iostream>
#include <kv/defint.hpp>

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";
}
```

精度保証付き数値積分の例 (2/3)

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

kv-0.4.41	[38.38352626453522 7,38.38352626464969]
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)	38.248589488377 54677984
keisan (Gauss-Legendre)	116.44815670772 5851273
intde2 by ooura	32.4641
python + scipy	36.48985372847387
CASIO fx-5800P	38.38352669

精度保証付き数値積分の例 (3/3)



常微分方程式の初期値問題

一階連立常微分方程式

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

初期値問題の精度保証アルゴリズム

$t_0 < t_1 < t_2 < \ldots$ に対して、

- ベキ級数演算 (PSA) を用いた、$x(t_i)$ を元に $x(t_{i+1})$ を精度保証付きで計算する方法 (短い区間での精度保証)
- Affine Arithmetic を用いて区間幅の膨らみを抑制しながら、短い区間での精度保証で得られた解を長い区間に渡って接続する方法

短い区間での精度保証 (1/3)

$v = x(t_s)$ の値を初期値として、$x(t_e)$ の値を計算する。

平行移動 & 両辺を積分で不動点形式に

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

解の Taylor 展開の生成

Type-I PSA 型の変数 $X_0 = v, T = t$ を用いて、$k = 0$ とし、

1. 次数 k の Type-I PSA で以下を計算

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

2. 次数 $k = k + 1$ とする。

を n 回繰り返すと、X_n として解の n 次の Taylor 展開が得られる。

短い区間での精度保証 (2/3)

解の存在保証

Type-II PSA の定義域を $D = [0, t_e - t_s]$ と設定し、Type-I PSA の反復で得られた n 次の Taylor 近似

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

と $T = t$ を用いて、

1. X_n の最終項の係数を膨らませた候補者集合

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

   を作成する。
2. $v + \int_0^t f(Y_c, T + t_s) dt$ を次数 n の Type-II PSA で計算し、$n+1$ 次から n 次に減次したものを

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

   とする。$n-1$ 次までの係数は X_n と全く同じになることに注意。
3. $V \subset V_c$ なら Y 内に解の存在が保証される。

短い区間での精度保証 (3/3)

解を含みそうな候補者集合の作成は、例えば次のように行なうことが出来る。

候補者集合の作成

1. $v + \int_0^t f(X_n, T + t_s)dt$ を次数 n の Type-II PSA で計算し、$n+1$ 次から n 次に減次したものを $Y_0 = x_0 + x_1 t + \cdots + V_0 t^n$ とする。
2. $r = \|V_0 - x_n\|$ とし、

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

とする。(半径を Newton 法の修正量の 2 倍にする。)

短い区間での精度保証の例

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

展開の次数: $n = 2$
10 進 3 桁演算。

(Type-I PSA による Taylor 展開の生成)

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

(候補者集合の生成)

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

2 次に減次して、

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

$r = \|[0.9, 1] - 1\| = 0.1$ なので、

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

(Type-II PSA による精度保証)

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

2 次に減次して、

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

$[0.886, 1] \subset [0.8, 1.2]$ なので、Y 内に真の解が存在する。

長い区間に渡って接続する (1/2)

flow map

常微分方程式において、値 $x(t_s)$ ($t = t_s$ における初期値) に対して値 $x(t_e)$ (時刻 $t = t_e$ における解の値) を対応させる写像

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

を flow map と呼ぶ。

初期値に関する変分方程式

$x^*(t)$ を v を初期値とした常微分方程式の解とすると、

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

を解くことによって、flow map の微分 (ヤコビ行列) を $\phi'_{t_s,t_e}(v) = y(t_e)$ で得られる。

長い区間に渡って接続する (2/2)

時刻 $t_0 < t_1 < t_2 < \cdots$ における解の包含を J_i とする。

平均値形式

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

- 単純にこのまま計算すると、wrapping effect によって区間幅が激しく増大する。
- wrapping effect を抑えるため、affine arithmetic を使って接続する

ステップ幅 Δt の決定

ε_0 を 1 ステップで混入する誤差の目標値とする。例えば machine epsilon。

1. Type-I PSA を用いて Taylor 展開を計算し、その係数を見て適切なステップ幅 Δt_0 を推定する。Type-I PSA で計算された Taylor 展開を

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

として、

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

とする。
2. ステップ幅 Δt_0 を用いて Type-II PSA を使って候補者集合を作成する。
3. その候補者集合を使ったとき新たに混入する誤差を ε として、新しいステップ幅を

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

で推定する。ただし、n は Taylor 展開の次数。
4. ステップ幅 Δt_1 を用いて Type-II PSA を使って再度候補者集合を作成し、解の存在検証を行う。検証に失敗したら例えばステップ幅を半分にする。

常微分方程式の初期値問題の例 (1/4)

van del Pol 方程式

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

一階に直す

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

常微分方程式の初期値問題の例 (2/4)

```
#include <kv/ode-maffine.hpp>
namespace ub = boost::numeric::ublas;
typedef kv::interval<double> itv;
class VDP { // 解きたい問題の右辺を関数オブジェクトで記述
    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.; // 初期値
    x(1) = 1;
    end = 100.; // 終了時刻
    kv::odelong_maffine(VDP(), x, itv(0.), end); // 初期値問題を解く(0-end)
    std::cout << x << "\n";
}
```

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

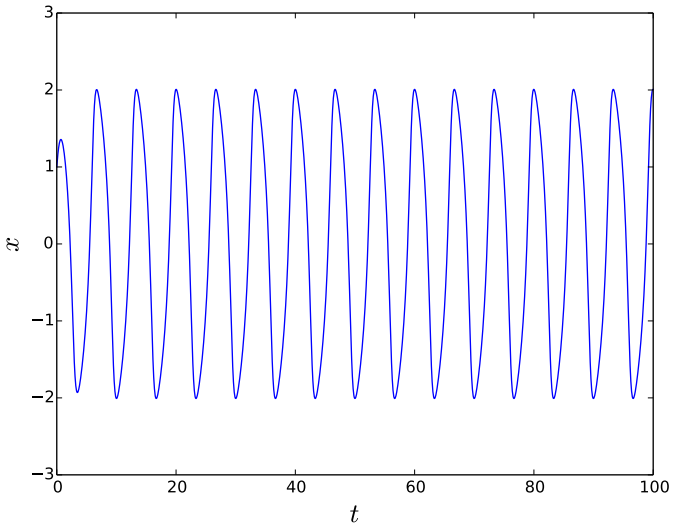
常微分方程式の初期値問題の例 (3/4) (途中経過の表示)

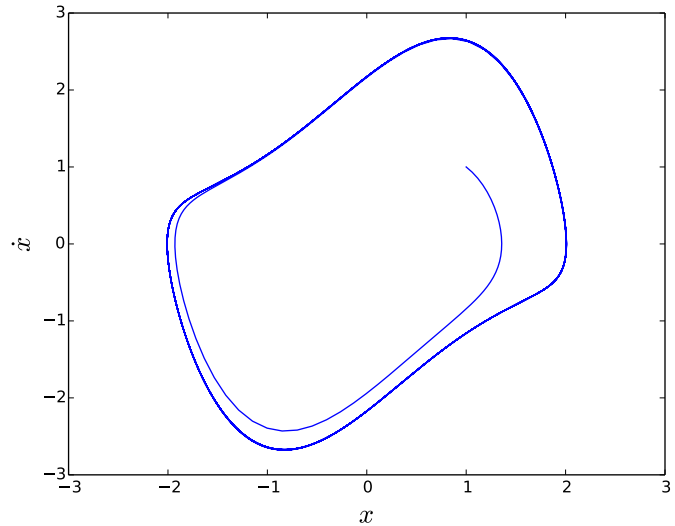
2^{-4} 間隔で密出力させた例

```
#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))));
}
```

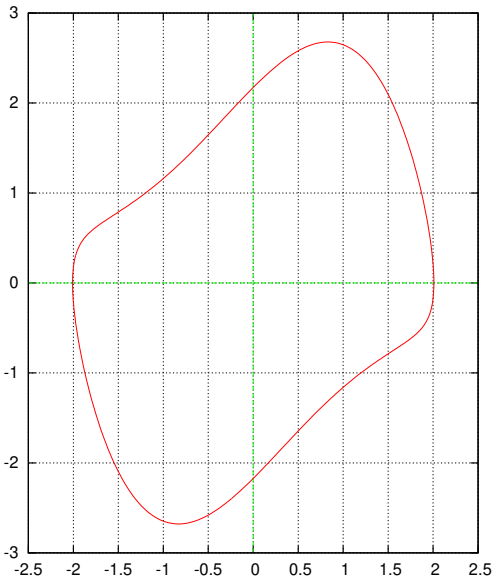
常微分方程式の初期値問題の例 (4/4) (途中経過の表示)

```
t: [−0,0]
[2]([1,1],[1,1])
t: [0.0625,0.0625]
[2]([1.0604282381493324,1.0604282381493327],[0.9318643053999509,0.9318643053999521])
t: [0.125,0.125]
[2]([1.1162696582692208,1.1162696582692211],[0.8534995323034189,0.8534995323034190])
t: [0.1875,0.1875]
[2]([1.1669406889500346,1.1669406889500352],[0.7667434979600858,0.7667434979600859])
t: [0.25,0.25]
[2]([1.211981145751376,1.2119811457513768],[0.6736807111275599,0.6736807111275599])
————————————omitted————————————
t: [99.9375,99.9375]
[2]([2.0074651477352984,2.0074651477354078],[0.07045240241356479,0.07045240241453340])
t: [100,100]
[2]([2.0077904809520377,2.007790480952215],[−0.0560514387514576,−0.0560514387502554])
```





境界値問題の例 (1/2)



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

$\phi_{0,p}$: 時刻 0 から時刻 p への flow map

p : 周期

s : Poincaré 断面の式

未知数は(x, y, p)

van der Pol 方程式 ($\mu = 1$) の周期解を Poincaré Map に対する Krawczyk 法で精度保証した例 (Poincaré 断面は $x = 0$)。周期は、

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

境界値問題の例 (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、5 つの解の存在保証に成功した。

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	

2021 年ノーベル物理学賞

THE NOBEL PRIZE

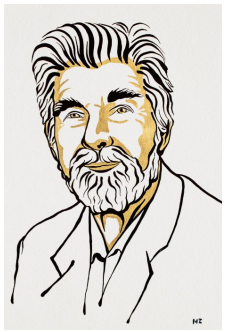
The Nobel Prize in Physics 2021 | Explore

The Nobel Prize in Physics 2021

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

KUNGL. VETENSKAPS-AKADEMIEN
THE ROYAL SWEDISH ACADEMY OF SCIENCES

5 OCTOBER 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

THE ROYAL SWEDISH ACADEMY OF SCIENCES has as its aim to promote the sciences and strengthen their influence in society.

BOX 50005 (LILLA FRESCATIVÄGEN 4 A), SE-104 05 STOCKHOLM, SWEDEN
TEL +46 8 673 95 00, KVA@KVA.SE • WWW.KVA.SE

Scientific Backgroud 文書は · · ·

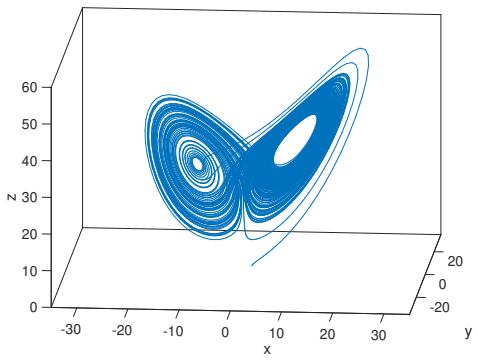
複雑系の元祖として、Lorenz 方程式の初期値鋭敏性の説明から始まる。

upper and lower boundaries [97]. The model is

$$\frac{dX}{dt} = \sigma(Y - X),$$
$$\frac{dY}{dt} = X(Ra - Z) - Y \qquad \text{and}$$
$$\frac{dZ}{dt} = XY - \beta Z,$$

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).



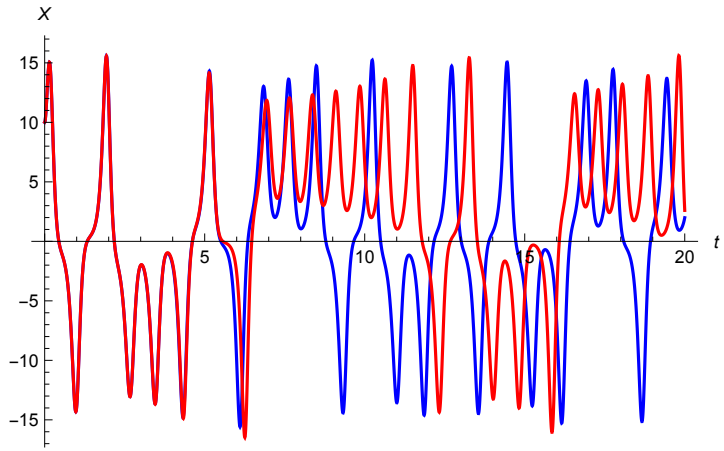
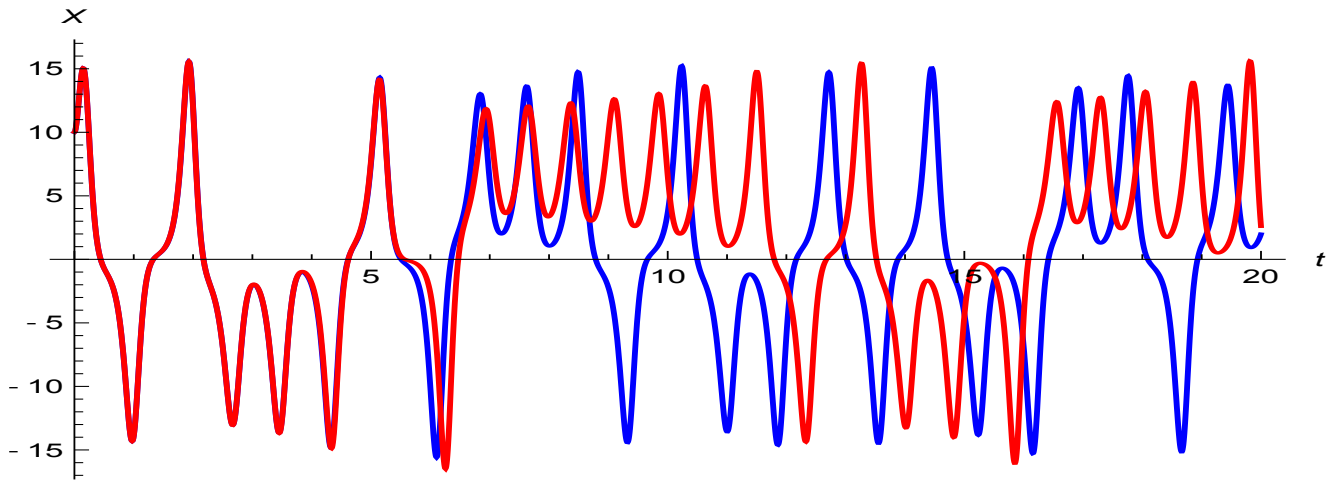


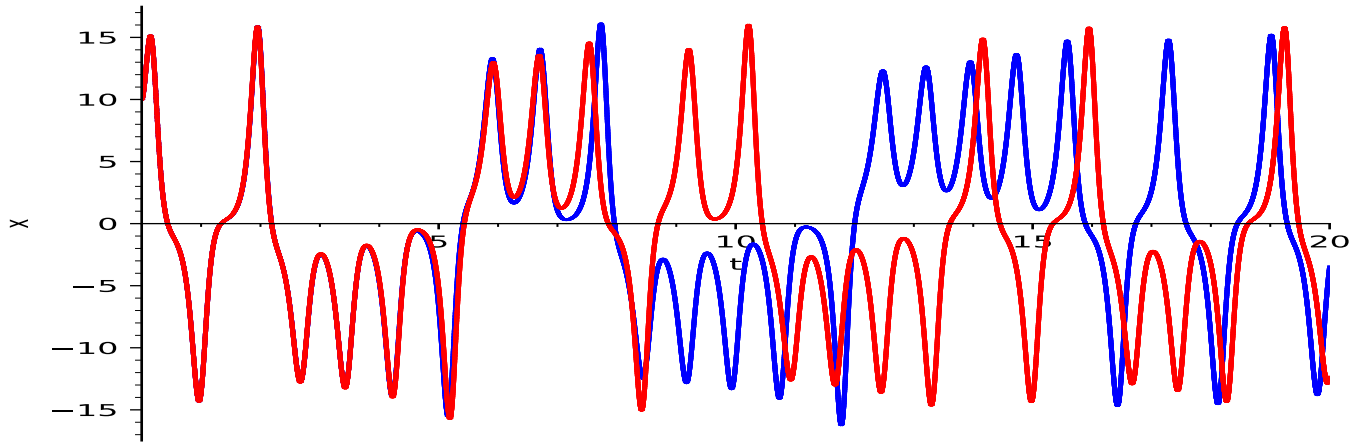
FIG. 2. Plot of $X(t)$ of the Lorenz system with $(\sigma, \beta, Ra) = (10, 8/3, 24.9)$ in which the initial data for all three variables are 10 (blue) or 10.01 (red). The divergence of the two solutions with slightly different initial conditions begins at $t = 5.5$; this is sensitive dependence on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

初期値 10 の場合の青いグラフと、初期値 10.01 の場合の赤いグラフで途中から全然違った軌道になる。いわゆる Buttefly Effect。

試しに精度保証付きで再計算してみたら・・・



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



(Verified Solution of Lorenz Equation calculated by kv library)

初期値の違いによる誤差よりも数値計算の誤差の方がずっと大きい！

まとめ

kv ライブラリ

- http://verifiedby.me/kv/ で公開中。
- 言語は C++。boost C++ Libraries も必要。
- 全てヘッダファイルで記述されており、インストールはヘッダファイルをどこかに配置するだけ。
- オープンソースである。精度保証付き数値計算の結果が「証明」であると主張するならば、計算に使われたプログラムは必ず公開されているべき。
- 計算に使う数値の型は double に制限されていない。C++のテンプレート機能を用いて容易に変更することが出来る。
- (数値型) 区間演算 (多数の数学関数含む)、4 倍精度 (double-double) 演算、MPFR ラッパー、複素数演算、自動微分、affine arithmeric、ベキ級数演算、とそれらの組み合わせ。
- (アプリケーション) Krawczyk 法による非線形方程式の精度保証、非線形方程式の全解探索、常微分方程式の初期値問題、常微分方程式の境界値問題、数値積分、特殊関数、他

皆様のご利用をお待ちしております!