精度保証付き数値計算プログラムの実装について

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

三部会連携 応用数理セミナー
(2015 年 12 月 24 日)

発表の概要

- 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

精度保証付き数値計算

精度保証付き数値計算

計算をすると同時に計算結果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 (解の存在する十分条件を区間演算で確認して、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)

区間演算

- 精度保証付き数値計算の基本技術。
- 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 標準でも定義されている「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)

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

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

- 除算 $X / Y =$

	$Y < 0$	$Y > 0$
$X < 0$	$[b \underline{/} c, a \overline{/} d]$	$[a \underline{/} c, b \overline{/} d]$
$X \ni 0$	$[b \underline{/} d, a \overline{/} d]$	$[a \underline{/} c, b \overline{/} c]$
$X > 0$	$[b \underline{/} d, a \overline{/} c]$	$[a \underline{/} d, b \overline{/} c]$

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

区間演算 (3)

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` を使えば簡単になった。

```
#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 (4)

実装上の注意点

- 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。
- 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。
- 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ので、自作する必要がある。

区間演算 (5)

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

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。

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。

$f'(I)$ (区間 I におけるヤコビ行列を包含する区間行列) が必要 $\Longrightarrow$ 自動微分の必要性

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。
- 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。
- 計算に使う数値の型を double 以外の型に容易に変更することが出来る。
- 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、精度保証付き数値計算のプログラムが公開されていないという状態はありえない。

最終更新: 2015/11/26

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。)。

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 区間演算(interval) の項を参照。

また、4倍精度演算 を行うときにIntelのFPUは いろいろ問題を抱えており、一応対策しているが、特に32bitモードでは問題が 発生する可能性があることを注意しておく。

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

3. ダウンロードとインストール

ダウンロード: kv-0.4.27.tar.gz (2015年11月26日公開)
(古いversionはこちら)

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。) また、IntelのCPUで64bit OSの場合は、-DKV_FASTROUND を付けると区間演算が速くなる。

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. MPFRラッパー

8. 複素数演算(complex)

9. 自動微分(autodif)

10. Affine Arithmetic (affine)

11. べキ級数演算 (psa)

12. Krawczyk法による非線形方程式の解の精度保証

13. 非線形方程式の全解探索

14. 常微分方程式の初期値問題の精度保証

15. 初期値問題solverと射撃法による境界値問題の精度保証

16. 数値積分

17. 特殊関数の精度保証

18. その他の機能

19. 関数オブジェクトによる問題の記述

20. その他

20.1 boostとは

20.2 行列計算 (boost.ublas)

20.3 数値型のプログラミング

21. 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。

22. 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。

(2015年10月5日追記) このソフトウェアはMITライセンスに基づいて公開されています。

更新履歴

開発にご協力下さった方々

kv - C++ Numerical Verification Libraries by kashi / kashi@waseda.jp

KV ライブラリの主な機能

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

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

なぜ C++を選んだか?

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

- 同じ表記の数式 (プログラム) に対して、
  - 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「数値型」を「流し込む」ことが多い。
- 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

```cpp
#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);
    std::cout << a << " " << b << "\n";

    double x=1., y=2.;

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

テンプレートあり

```cpp
#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);
    std::cout << a << " " << b << "\n";

    double x=1., y=2.;

    swap(x, y);
    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 の場合のより高速なオプションや、丸めの変更を全く行わないオプションもある。

区間演算プログラムの例

```
#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";
}
```

```
[7.485470860549956,7.4854708605508238]
```

使い方

解凍

```
$ ls
kv-0.4.27.tar.gz
$ tar xfz kv-0.4.27.tar.gz
$ ls
kv-0.4.27/  kv-0.4.27.tar.gz
$ cd kv-0.4.27
$ ls
LICENSE.txt  README.txt  example  kv  test
```

必要なのは kv ディレクトリ以下。適当な場所に配置する。

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)

```
#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";
}
```

```
[7.485470860550344912656518204308257,7.485470860550344912656518204360964]
```

Krawczyk 法による解の精度保証

```
#include <kv/kraw-approx.hpp>
namespace ub = boost::numeric::ublas;
struct Func {
    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])
```

全解探索の例

```
#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]([0.25115183522076645,0.25115183522076507],[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))
```

double-double(dd)

- いわゆる `twosum` と `twoproduct` を用いた擬似 4 倍精度演算。
- 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。

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 $-1 \le \varepsilon_i \le 1$ 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。

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。

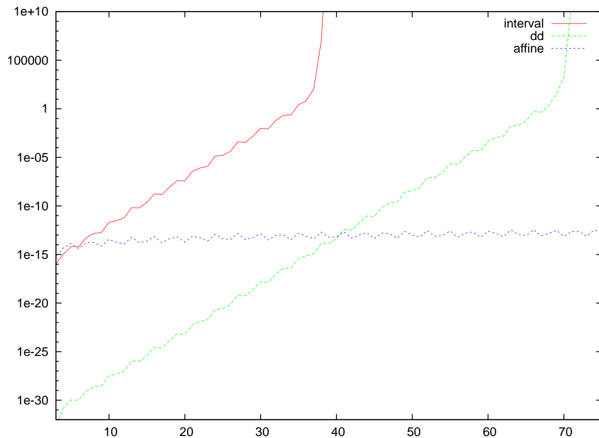
```cpp
#include <kv/interval.hpp>
#include <kv/rdouble.hpp>
int main()
{
    int i;
    kv::interval<double> x, y, z;
    std::cout.precision(17);
    x = 1.;
    y = 1.;
    for (i=2; i<=10000; i++) {
        z = (1 + 2 * y) / (x * y * y);
        std::cout << i << " " << z << "\n";
        x = y;
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

```cpp
#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

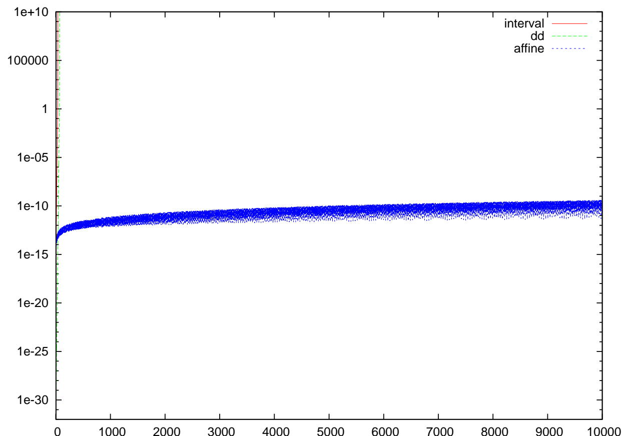
n	区間演算	Affine Arithmetic
2	$[3, 3]$	$[3, 3]$
3	$[0.7777777777777767, 0.7777777777777778]$	$[0.7777777777777756, 0.7777777777777824]$
4	$[1.408163265306122, 1.408163265306123 2]$	$[1.4081632653061197, 1.4081632653061247]$
5	$[2.474480151228730 2, 2.474480151228736 9]$	$[2.4744801512287271, 2.4744801512287414]$
6	$[0.6899539592210210 9, 0.6899539592210273 2]$	$[0.6899539592210222, 0.6899539592210262 1]$
7	$[2.020393474742363, 2.0203934747424169]$	$[2.0203934747423817, 2.0203934747423987]$
8	$[1.7898074714307314, 1.7898074714308816]$	$[1.7898074714307978, 1.7898074714308153]$
$\vdots$	$\vdots$	$\vdots$
31	$[0.7009891618227720 4, 0.7094198209793560 8]$	$[0.7051917561686529 2, 0.7051917561686842 4]$
32	$[1.7816188152293368, 1.8444838202503787]$	$[1.8127715215496742, 1.8127715215497711]$
33	$[1.890688867011997, 2.1073484458445711]$	$[1.9960405520559754, 1.9960405520560838]$
34	$[0.5837212479498864 4, 0.8187930456850460 8]$	$[0.6911938131215669 1, 0.6911938131216002 3]$
35	$[1.5341327531940911, 4.0942614522583929]$	$[2.4982930525184534, 2.4982930525186054]$
36	$[0.2964039576199632 9, 6.6882779916662063]$	$[1.3900085495715059, 1.390008549571586]$
37	$[0.0086967943592607538, 106.66548725824453]$	$[0.7830967853484550 6, 0.7830967853484963 7]$
38	$[1.3369859317919986 \times 10^{-5}, 9560542.5436595381]$	$[3.0105168251706007, 3.0105168251708015]$
39	$[1.0257053348148149 \times 10^{-16}, 1.229984619387229 \times 10^{19}]$	$[0.9892441618081190 2, 0.9892441618081792 1]$
40	$[6.9138205018986439 \times 10^{-46}, 1.7488703313159241 \times 10^{56}]$	$[1.0109923081577889, 1.0109923081578503]$
41	$[2.6581843623384974 \times 10^{-132}, 7.1339291414655989 \times 10^{162}]$	$[2.9887738208443805, 2.9887738208445911]$
42	$[0, \infty]$	$[0.772625180482649 6, 0.7726251804826975 8]$
43	—	$[1.4265918581977079, 1.4265918581978075]$
$\vdots$	$\vdots$	$\vdots$
9999	—	$[0.7607151065993281 7, 0.7607151066789953 4]$
10000	—	$[1.4727965248961243, 1.4727965251850226]$

区間幅のグラフ



- n と区間幅の関係。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)



- n と区間幅の関係。($n = 10000$ まで)

ベキ級数演算 (psa)

- ベキ級数演算のためのライブラリ。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。
- 高次の項を捨ててしまう Type-I PSA と、高次の項を捨てずに区間係数として残す Type-II PSA がある。
- Type-II PSA では、最高次の項の係数のみを (幅の広い) 区間とし、それ以外の低次の項の係数は点区間 (あるいは丸め誤差程度の幅の狭い区間) とする点に特徴がある。

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 $=[0,0.1]$
$1+t-4t^2$	$1+t+[-4,-3.5]t^2$

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

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や数学関数を考える。
- 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。
- 1 変数関数の高階微分を計算する自動微分法とほとんど同じ。

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)

$$
\begin{aligned}
x(t) &= x_0 + x_1 t + x_2 t^2 + \cdots + x_n t^n \\
y(t) &= y_0 + y_1 t + y_2 t^2 + \cdots + y_n t^n
\end{aligned}
$$

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

$$
\begin{aligned}
x(t) &= 1 + 2t - 3t^2 \\
y(t) &= 1 - t + t^2
\end{aligned}
$$

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

Type-II PSA の演算規則 (2)

乗算

(1) まず、打ち切り無しで乗算を行う。

$$
\begin{aligned}
x(t) \times y(t) &= z_0 + z_1 t + \cdots + z_{2n} t^{2n} \\
z_k &= \sum_{i=\max(0,k-n)}^{\min(k,n)} x_i y_{k-i}
\end{aligned}
$$

(2) $2n$ 次から n 次に減次する。

m 次から n 次への減次

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

$$
\begin{aligned}
z_i &= x_i \quad (0 \le i \le n-1) \\
z_n &= \left\{ \sum_{i=n}^{m} x_i t^{i-n} \mid t \in D \right\}
\end{aligned}
$$

Type-II PSA の演算規則 (3)

乗算の例

定義域を $D = [0, 0.1]$ とする。

$$\begin{aligned} x(t) &= 1 + 2t - 3t^2 \\ y(t) &= 1 - t + t^2 \end{aligned}$$

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

Type-II PSA の演算規則 (4)

sin などの数学関数

その関数を g として、

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

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)

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

積分区間内に特異点を持たない数値積分の方法を示す。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 (特異点無し) の例 (1)

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

使用プログラム

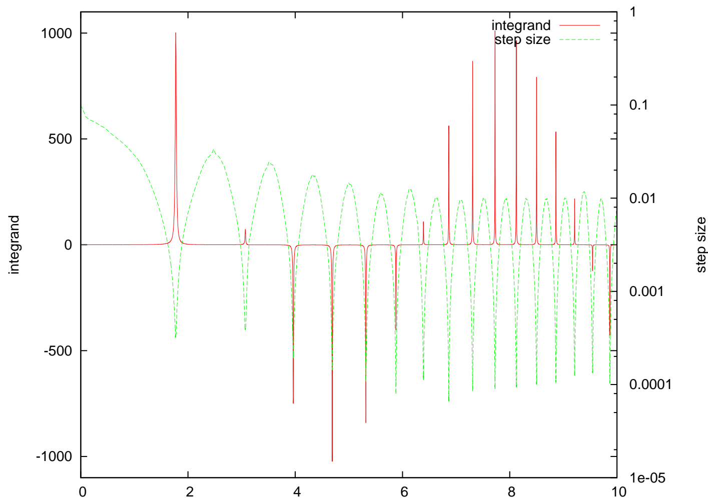
```
#include <iostream>
#include <kv/defint.hpp>

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 (特異点無し) の例 (2)

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

kv-0.4.23	[38.383526264535703,38.383526264649654]
octave 3.8.1	38.3837105761501
intlab 9	[38.34845927756175, 38.41859325162576]
Mathematica 10.1.0	0.0608979
matlab 2007b	38.383519835854528
keisan (ロンバーグ)	38.324147930794
keisan (Tanh-Sinh)	38.24858948837754677984
keisan (ガウス-ルジャンドル)	116.448156707725851273
intde2 by ooura	32.4641
python + scipy	36.48985372847387
CASIO fx-5800P	38.38352669

精度保証付き数値積分 (特異点無し) の例 (3)



常微分方程式の初期値問題

一階連立常微分方程式

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

初期値問題の精度保証アルゴリズムは、$t_0 < t_1 < t_2 < \dots$ に対して、

- $x(t_i)$ 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)
- 短い区間での精度保証法を利用して、長い区間に渡って (区間幅の膨らみを抑制しながら) 接続する方法

の 2 つに分けて考えることが出来る。

短い区間での精度保証 (1)

平行移動 & 両辺を積分で不動点形式に

$$x(t) = v + \int_0^t f(x(t), t + t_i)dt$$
$$(v = x(t_i), \quad t \in [0, t_{i+1} - t_i])$$

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

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

(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)

解の存在保証

Type-II PSA の定義域を $D = [0, t_{i+1} - t_i]$ 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

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

を作成する。

(2) $v + \int_0^t f(Y, T + t_s) dt$ 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

$$Y_1 = x_0 + x_1 t + x_2 t^2 + \cdots + V_1 t^n$$

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) $V_1 \subset V$ なら Y_1 内に解の存在が保証される。

短い区間での精度保証 (3)

候補者集合の作成

候補者集合の作成は、例えば次の手順で行う。

(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、

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

とする。

短い区間での精度保証例

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

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)

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

(候補者集合の生成)

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

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、

$$Y_0 = \boxed{1 - t + [0.8, 1.2]t^2}$$

(Type-II PSA による精度保証)

$$\begin{aligned}& 1 + \int_0^t (-Y_0^2)\mathrm{d}t \\ =& 1 - t + t^2 + [-1.133, -0.786]t^3\end{aligned}$$

2 次に減次して、

$$Y_1 = \boxed{1 - t + [0.886, 1]t^2}$$

$[0.886, 1] \subset [0.8, 1.2]$ なので、Y_1 内に真の解が存在する。

長い区間に渡って接続する (1)

推進写像

$t = t_s$ における値 $v = x(t_s)$ に対して、$x(t_e)$ を対応させる写像

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

を推進写像と呼ぶことにする。

初期値に関する変分方程式

$x^*(t)$ を v を初期値とした与式の解とすると、

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

を解くことによって、推進写像の微分を $\phi'_{t_s,t_e}(v) = y(t_e)$ で計算出来る。

長い区間に渡って接続する (2)

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

平均値形式

$$X_{i+1} = \phi_{t_i,t_{i+1}}(\mathrm{mid}(X_i)) + \phi'_{t_i,t_{i+1}}(X_i)(X_i - \mathrm{mid}(X_i))$$

- 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。
- affine arithmetic を使って接続する
- ステップ幅の調整は、数値積分の場合と同じアルゴリズムを使う。

例題: van der Pol 方程式

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

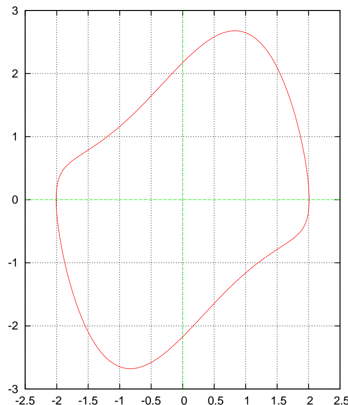
$$\begin{aligned} \frac{dx}{dt} &= y \\ \frac{dy}{dt} &= \mu(1 - x^2)y - x \end{aligned}$$

常微分方程式の初期値問題の例

```
#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
    odelong_maffine(VDP(), x, itv(0.), end); // 初期値問題を解く(0-end)
    std::cout << x << "\n";
}
```

```
[2]([2.0077904809521123,2.0077904809521399],[-0.056051438751190147,-0.056051438750530216])
```

境界値問題の例 (1)



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。周期は、

$$T = [6.6632868593231044, 6.6632868593231534]$$

境界値問題の例 (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])
```

まとめ (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。
- 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。
- 計算に使う数値の型を double 以外の型に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
- (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
- 皆様のご利用をお待ちしております!