常微分方程式の精度保証パッケージ開発

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

早稲田大学

第 1 回山梨精度保証研究会 (2014 年 9 月 15 日)

発表の概要

- ライブラリの紹介
- 常微分方程式の精度保証概要
- ポアンカレマップの精度保証について

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。

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

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、実行時には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)

```
#include <iostream>

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

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

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

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

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

    swap_int(a, b);
    std::cout << a << " " << b << "\n";

    double x=1., y=2.;

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

```
#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)

```
#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();
}
```

```
#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

- 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 ライブラリは現状では大きな行列を扱うと遅い。

区間演算

- 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。
- 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-0.4.8.tar.gz
$ tar xfz kv-0.4.8.tar.gz
$ ls
kv-0.4.8/  kv-0.4.8.tar.gz
$ cd kv-0.4.8
$ ls
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";
}
```

```
[7.48547086055034491265651820430825 7,7.48547086055034491265651820436096 4]
```

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]([0.25115183522076507,0.25115183522076507],[1.2511518352207642,1.2511518352207654])(ex:
    improved)
ne_test: 49, ex_test: 3, ne: 23, ex: 2, giveup: 0
```

常微分方程式の初期値問題

一階連立常微分方程式

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

初期値問題の精度保証アルゴリズムは、$t_0 < t_1 < t_2 < \ldots$ 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)

Taylor 展開と剰余項の評価を用いる。詳細は省略。

- Lohner 法
- ベキ級数演算を使った方法
- その他いろいろ

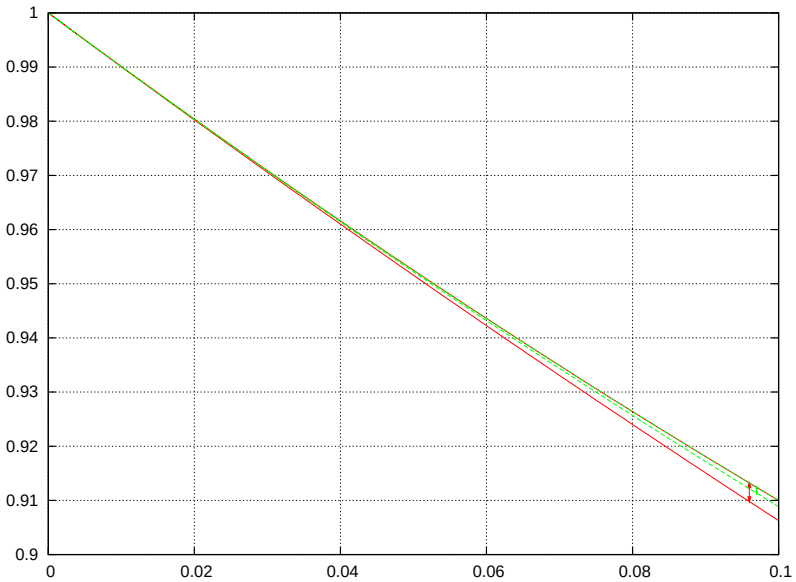
例えば以下の式の解を次数 2 で精度保証すると

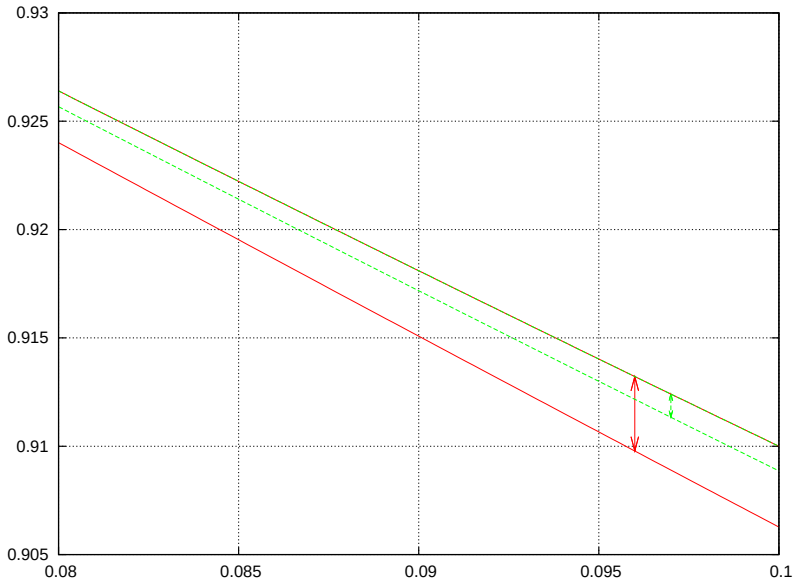
$$\frac{dx}{dt} = -x^2, \quad x(0) = 1, \quad x \in [0, 0.1]$$

Lohner 法	ベキ級数演算
$1 - t + [0.627, 1]t^2$	$1 - t + [0.886, 1]t^2$

のような区間多項式になる。

短い区間での精度保証 (2)





本発表の数値例ではベキ級数演算法を用いた。

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

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

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、

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

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。
- Lohner は、QR 分解を用いることによって wrapping effect を抑制した。
- affine arithmetic ならどうか?

長い区間に渡って接続する (3) (QR 分解による方法)

$c_0 = \mathrm{mid}(X_0)$, $Y_0 = X_0 - c_0$, $Q_0 = I$, $i = 0$ とし、

1. $c_{i+1} = \mathrm{mid}(\phi_{t_i, t_{i+1}}(c_i))$
2. $Y_{i+1} = (Q_{i+1}^{-1} \phi'_{t_i, t_{i+1}}(X_i) Q_i) Y_i + Q_{i+1}^{-1}(\phi_{t_i, t_{i+1}}(c_i) - c_{i+1})$
3. $X_{i+1} = Q_{i+1} Y_{i+1} + c_{i+1}$

を繰り返す。ただし、

- Q_{i+1} は $\mathrm{mid}(\phi'_{t_i, t_{i+1}}(X_i) Q_i)$ を QR 分解したものの Q とする (近似でよい)。
- Q_{i+1}^{-1} は Q_{i+1} の真の逆行列またはそれを含む区間行列でなければならない。

例題: van der Pol 方程式

van del Pol 方程式

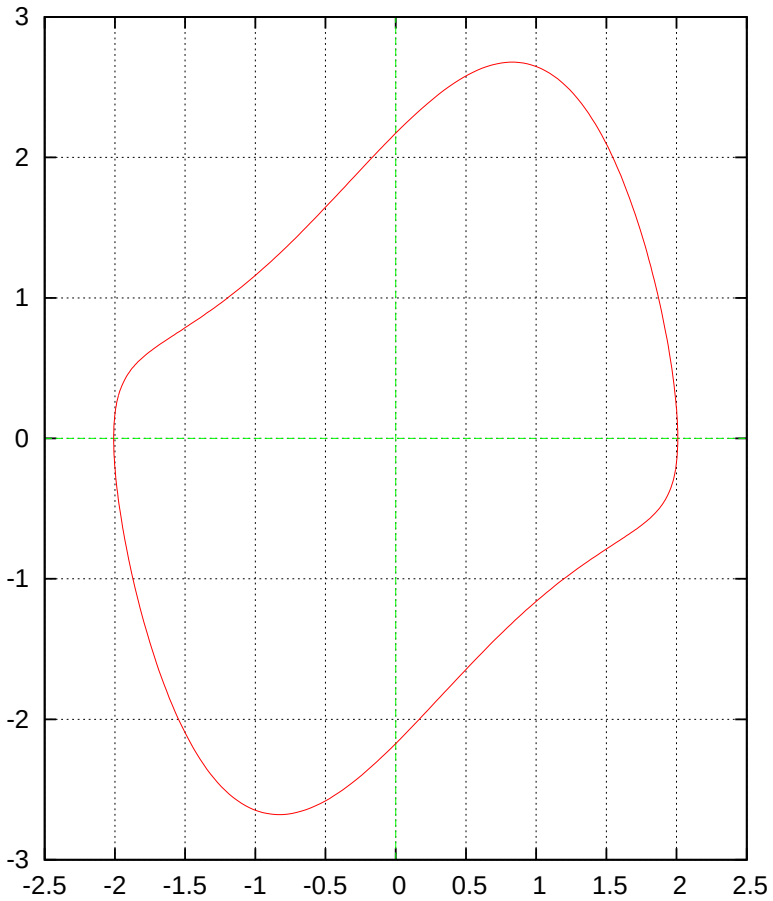
$$\frac{d^2x}{dt^2} - \mu(1-x^2)\frac{dx}{dt} + x = 0$$

一階に直す

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

- stiffness の強くない $\mu = 1$ で実験する。

リミットサイクル



試しにポアンカレマップを精度保証してみた ($\mu = 1$)。周期は、

$$T = [6.6632868593231071, 6.6632868593231543]$$

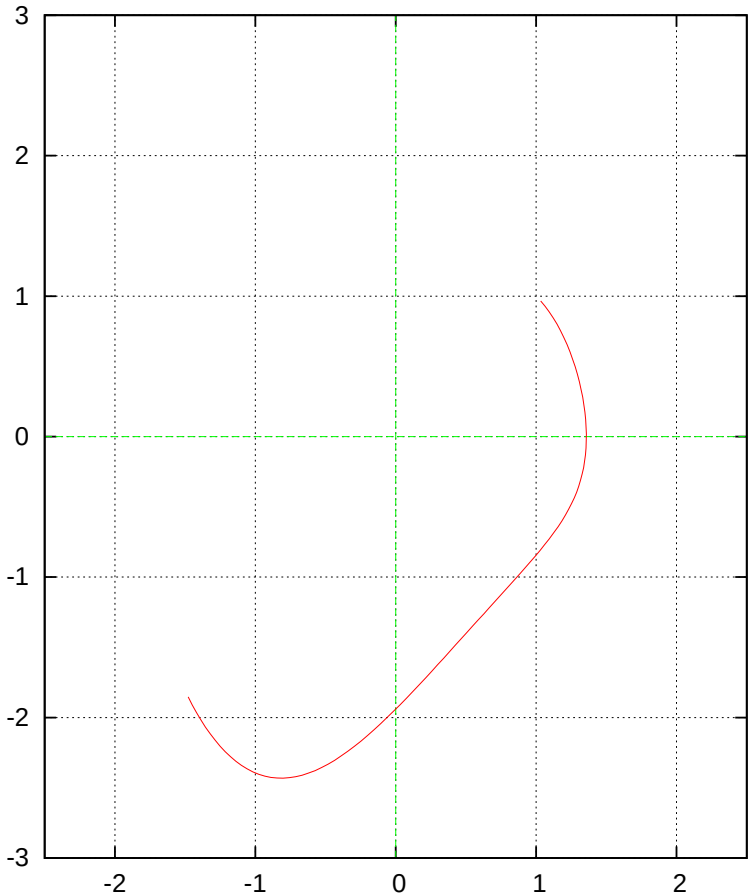
計算の詳細

- 初期値は $([1-10^{-4}, 1+10^{-4}], [1-10^{-4}, 1+10^{-4}])^T$
- 次数は 18 次
- 刻み幅は、$\varepsilon = 2^{-53}$、$n = 18$、解の taylor 展開を $a_0 + a_1 t + a_2 t^2 + \cdots$ として、

$$\delta = \frac{\sqrt[n]{|a_0|\varepsilon}}{\max(\sqrt[n-1]{|a_{n-1}|}, \sqrt[n]{|a_n|})}$$

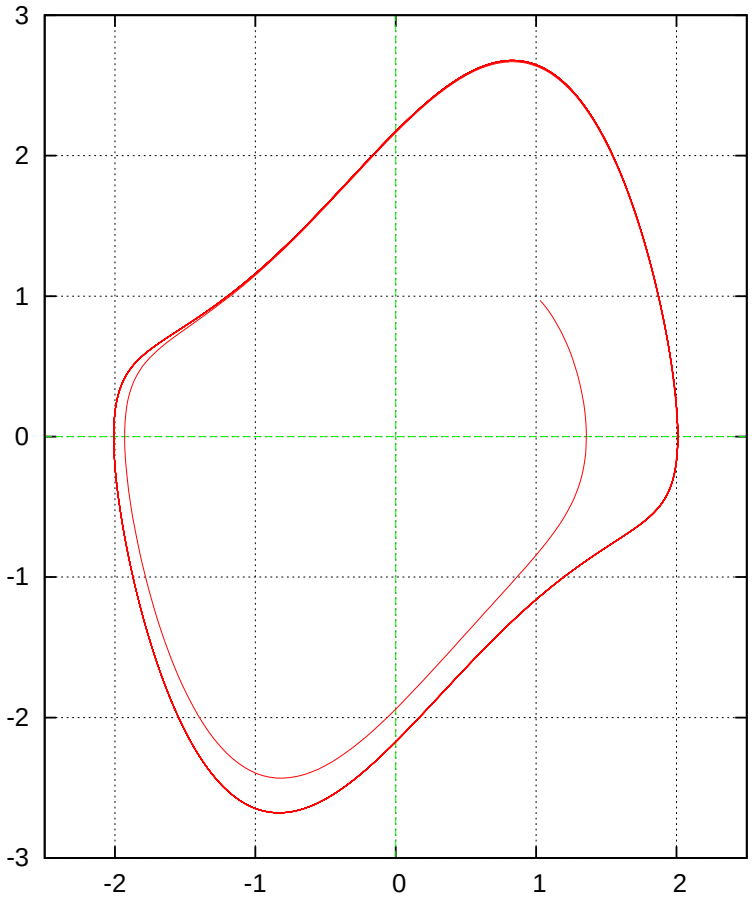
を目安として修正したものを使った。

計算結果 (1) (単なる区間演算)



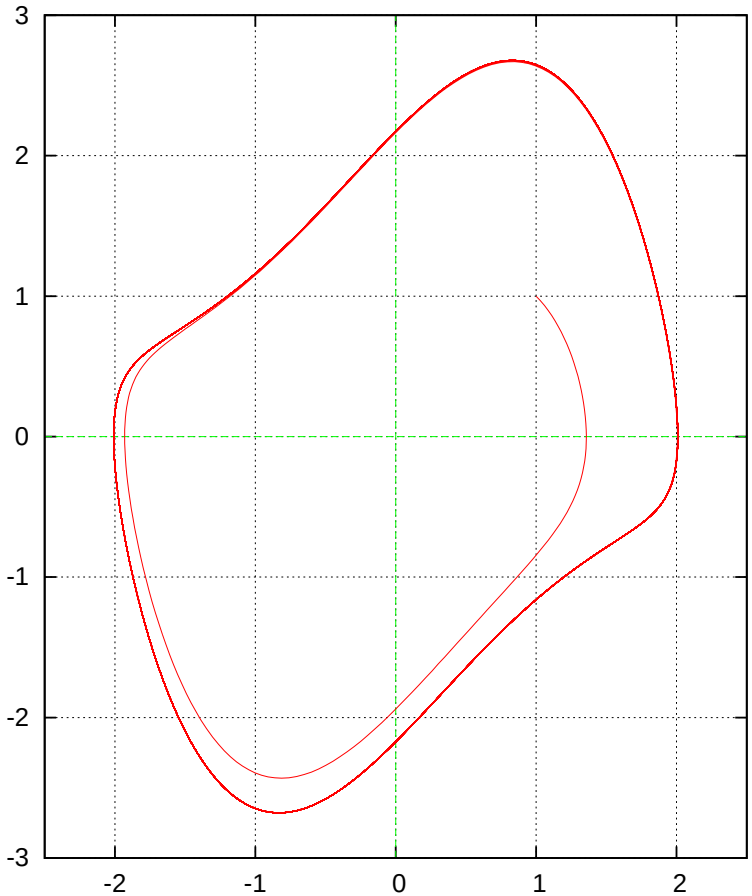
到達時刻は、 $t = 4.317$ (1 周も出来ない)

計算結果 (2) (QR 分解を使う)



到達時刻は、 $t = 176.31$ (約 26 周)

計算結果 (3) (affine arithmetic を使う)



到達時刻は、 $t = 1471.6$ (約 220 周)