C++言語で方程式を解く!
— GNU MP, MPFR による多倍長計算入門 —

静岡理工科大学 総合情報学部
コンピュータシステム学科
幸谷智紀
http://na-inet.jp/

2009 年 5 月 23 日 (土)

1 初めに

この資料は，コンピュータ言語や複素数になじみのない高校生向けに執筆されたものです。数学的な内容は複素係数の 2 次方程式の解を求めるだけのものですが，MPFR[3]/GMP[2] のクラスライブラリ gmpfrxx[4] を用いて多倍長計算も利用しているのがちょっと目新しいところでしょうか。

この資料を最初に使用したのは 2009 年 5 月 23 日 (土)10:00〜12:00, 13:00〜15:00 に，静岡理工科大学で実施された，スーパーサイエンスハイスクール受講者向けの実験講座でした。当初は下記のような流れで時間内に全て終わるように計画されていました。

1. 授業内容概説と実習環境の解説 (15 分)
2. C++言語の解説とプログラミング演習 (35 分)
3. 1 次方程式と 2 次方程式を解くプログラム作成 (50 分)
4. 複素数解を持つ 2 次方程式への対応 (25 分)
5. 多倍長精度化と数値誤差の導出 (25 分)
6. 数値実験とレポート作成 (50 分)

複素数にも有限桁の浮動小数点数演算にも馴染みのない受講生さん達には相当つらい内容だったかもしれません。全員が多倍長化まで辿り着くことはできませんでしたが，少なくとも実施した本人 (幸谷) は十分楽しんで本資料と実験を行っ

たことだけは確かなことです (はた迷惑ですね)。MPFR/GMP を使いこなすためのC++クラスライブラリはまだまだ未成熟なので，この資料で示した以外の計算をやろうとすると相当パッチを当てないと使い物にならないと思いますが，スタートダッシュぐらいは切れるレベルには達していると判断しています。本資料が，世界標準の多倍長計算ライブラリへの一歩になれば幸甚に思います。

2　C++言語とは？

C++(「シープラスプラス」と読みます。) はコンピュータ言語の一種です。

コンピュータ (Computer) は，その名の通り，計算 (compute) しかできません。それも，与えられた指示書，すなわちプログラム (program) に書いたとおりにしか動くことが出来ませんし，もしプログラムに書いてある以外の動作を行ったとすれば，それは単なる故障でしかありません。

そのプログラムに，コンピュータにやって欲しい仕事を順序立てて書く手段が，コンピュータ言語 (Computer Language) なのです。C++は，C 言語にオブジェクト指向 (object-oriented) な機能を追加した，比較的新しい言語体系ですが，昔から使われてきた C(言語) プログラムを見通しよくまとめるのに便利なこともあって，比較的たくさんの利用者を集めてきました。

今回はこの C++言語の機能を使って，2 次方程式，たとえば

$$3x^2 + 2x - 1 = 0$$

を「完璧に」解くことを実現します。ここで言う『完璧」とは，どんな係数が与えられても答え，即ち「解（かい)」を出すことが出来る，ということを意味します。従って，

$$x^2 + 1 = 0$$

であっても

$$-5x^2 - 1.1x + 0.11 = 0$$

であっても，解が理論的に存在するとあらかじめ分かっている方程式であれば，全ての解を求めるようにプログラムを作っていきます。

「2 次方程式なんて朝飯前さ！」と思っている人は，ちょっと考え違いをしています。大体，教科書に載っている方程式は，教科書を書いた人達が，文科省のお役人様のご指導に従って，皆さん方のレベルにあわせて解きやすいように設計してあるものばかりだからです。今回目指すのはそのようなお仕着せの方程式ではなく，この資料を書いた奴が「受講生に苦労させてあげよう」という配慮の元で作られたものが入っています。そこのところを肝に銘じて，以下の説明をよく読み，そして実際にプログラムを作って動作させて下さい。

3　何はともあれ動かしてみよう

C++に限らず，昔から，コンピュータ言語は目（ディスプレイを見つめつつ) と頭 (自分のどこが間違っているのか考えつつ）と手（プログラムを打ち込む）を使って覚えるものです。黙って教科書を眺めてサンプルプログラムを漫然と打ち込むだけでは何も習得できません。何をやっているか，自分で理解しながら読み進んで下さい。

まず，次の C++プログラムを「テキストエディタ」で打ち込んで，”hellow.cc” という名前で保存して下さい。便宜上，本資料中のプログラムには行番号 (行頭の 10:など) が付いていますが，それは除外して，コロン (:) より後ろから打ち込んで下さい。

```
 1: #include <iostream>
 2:
 3: using namespace::std;
 4:
 5: int main()
 6: {
 7:     cout << "Hellow, C++!" << endl;
 8:
 9:     return 0;
10: }
```

大変つまらないプログラムですが，小学校一年生の国語の教科書の最初が「アカイアカイアサヒガアカイ」[1]のと同じようなもので，コンピュータ言語を学ぶ人は，コンピュータに「こんにちは (Hellow)」と表示させることから始まることになっています。ここでもそれを作って動かしてみましょう。

”hellow.cc”プログラムは，まだ打ち込んだだけでは動作しません。これを**ソースプログラム**，あるいは単に**ソース**と呼びます。ソースをコンピュータ自身が解釈できる形式に書き換えるためには，**コンパイラ** (compiler) という道具を使います。ここでは GCC[1] というコンパイラセットにある g++という C++コンパイラを使います。プロンプト$の後に，”g++”に続けてソースプログラムのファイル名を与えてコンパイルします。

```
$ g++ hellow.cc
```

もしプログラムに間違い（エラー）がなければ，”a.out”という実行可能ファイルが生成されます。エラーがあれば「てめぇのプログラムはここが間違っている！」というエラーメッセージが表示されますので，指示通りに直して下さい。

[1]今は違ったっけ？

実行可能ファイルは”./”を頭につけて名前を指定すると，コンピュータが実行してくれます。

```
$ ./a.out
Hellow, C++!
```

ちゃんと画面に”Hellow, C++!”という文字が出ましたでしょうか？　もしおかしなことがあれば

あなたが間違っている

ので，もう一度

1. ”hellow.cc”の内容が正しいか？
2. コンパイルの仕方は正しいか？
3. 実行可能ファイルを正しく実行しているか？

の 3 点を確認して下さい。故障でもしていない限り，プログラムが動かないのは

あなたが間違っている

のです。

3.1　倍精度浮動小数点数データの扱い方

今度は，小数表記の数，即ち実数 (real number) を扱うためのプログラム”double.cc”を書いて実行してみます。下記のプログラムは，キーボードから数字を入力 (input) し，それをディスプレイに出力 (output) するだけの機能しかありません。

```
 1: #include <iostream>
 2: #include <iomanip>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:     double a;
 9:
10:     cout << "a = ";
11:     cin >> a;
12:
```

```
13:     cout << "input a = " << a << endl;
14:
15:     return 0;
16: }

$ g++ double.cc
$ ./a.out
a = 4.2       <-- キーボードから"4.2"と入力すると ･･･
input a = 4.2 <-- 4.2 が出力される
```

さて，ここでプログラムの機能を簡単に見ていくことにしましょう。先の”hellow.cc”も同じ構造をしているので，共通している所，違っている所を判断し,「なぜそう書くのか？」と常に意識しておいて下さい。

大まかに言うと，C++プログラムは

```
 1: #include <iostream>
 2: #include <iomanip>
 3:
 4: using namespace std;
 5:
 6: int main()
 7: {
 8:      ･･･ ここに実行したい内容を記述する ････
 9:
10:     return 0;
11: }
```

という構造をしています。7 行目と 11 行目の中括弧 {} の中に，実行したい指令を書き込んでいきます。保存しておきたいデータは変数 (variable) という名前，即ち変数名が付いた入れ物を用意し，そこに入れて保存しておく必要があります。

指令の書き方の大原則は次の 3 点です。

1. 文字列以外は全て半角アルファベットで記述すべし！
2. 変数は使う前に「この変数名をこのデータ用に使うぞ！」と宣言しておくべし！
3. 記述の最後にはセミコロン (;) を付けるべし！

これは必ず守って下さい。そのほか，便宜上やっておいた方がいいこと等もありますが，それは追々触れていくことにします。慣れないうちは,「言われたとおりにやってみる」ことも重要です。

さて，C++の細かいことを少しずつ見ていくことにしましょう。

1，2 行目は，実行したい指令に必要な機能を取り込む (include) ために書いています。キーボードから入力したい時には

```
cin >> 変数名;
```

とと書きます。”cin”というのがキーボードからの入力データを意味します。ちょうど，cin から変数にデータが流れ込んでいく感じですね。

画面に出力したいときには，入力とは逆に

```
cout << 変数名;
```

と書きます。変数から”cout”(ディスプレイ画面) にデータが流れていくイメージをつかんで下さい。

文字列を出力したいときには，二重引用符 (double quotation)"で囲んで

```
cout << "Hellow, C++!" << endl;
```

と書きます。最後の”endl”は改行を意味します。”<<”でつなげておけば，変数も文字列も，いくらでも一行にまとめて書くことが出来ます。従って

```
cout << "input a = " << a << endl;
```

と書くと，”input a = ”に続いて，変数 a に保存されているデータが出力されます。。

さて，データの入れ物である変数名ですが，半角アルファベットで宣言しておきます。小数データを保存するためには，**倍精度浮動小数点数** (double precision floatint-point number) を使いますので

```
double a;
```

と宣言します。複数の変数を一度に宣言したいときには，カンマ (,) で区切って

```
double a, b, c;
```

と書きます。カンマの後のスペースは入れても入れなくても実行には関係ありません。

10 行目では，整数ゼロ (0) を OS，即ちコンピュータを動かしている基本ソフトウェアに戻して (return)，実行を終了しています。正しくはメイン関数 (main) を終了している，ということになるのですが，メイン関数が終了することは，即ちこのプログラム自体が終了することを意味しています。

今,「関数 (function)」という言葉を使いました。プログラムを簡潔に書くためには必要不可欠なものなのですが，それについては後で触れることにします。

練習問題1

では，ここまで習ってきたC++プログラムの機能を使って，次のようなプログラム”keisan.cc”を作って下さい。

1. 変数 a, b をキーボードから入力
2. $a+b, a-b, ab, \frac{a}{b}$ を計算し，その結果を出力する

倍精度浮動小数点数の変数として a, b が宣言されているとき，計算式は次のように書けます。

和 $\Longrightarrow$ `a + b`

差 $\Longrightarrow$ `a - b`

積 $\Longrightarrow$ `a * b`

商 $\Longrightarrow$ `a / b`

ではチャレンジしてみて下さい。

4　1次方程式を解く

さて，では今まで習ってきた機能を使って，一次方程式

$$ax + b = 0 \tag{1}$$

を解くプログラム”linear.cc”を作ってみましょう。ここで a, b は実数の定数を意味します。

このプログラムは次のような機能が必要です。

1. 一次方程式 (1) の係数 a, b の入力
2. $a = 0$ の時は計算しない！(答えがないので)
3. $a \neq 0$ の時は $x = -b/a$ を出力

これをC++プログラムにしたのが下記の”linear.cc”です。

```
1: #include <iostream>
2: #include <iomanip>
3:
4: using namespace std;
```

```
 5:
 6: int main()
 7: {
 8:     double a, b, x;
 9:
10:     cout << "a = ";
11:     cin >> a;
12:     cout << "b = ";
13:     cin >> b;
14:
15:     cout << "linear equation: " << a << " * x + " << b << " = 0"
 << endl;
16:
17:     if(a == 0.0)
18:     {
19:         cout << "Cannot solve!" << endl;
20:         return -1;
21:     }
22:
23:     x = -b / a;
24:
25:     cout << "x = " << x << endl;
26:
27:     return 0;
28: }
```

ではコンパイルして実行していきましょう。まずは,

$$-3x + 3.1 = 0$$

を解きます。答えは

$$x = \frac{3.1}{3} = 1.0333\cdots$$

となりますね。これを確認して下さい。

```
$ g++ linear.cc
$ ./a.out
a = -3
b = 3.1
linear equation: -3 * x + 3.1 = 0
x = 1.03333
```

あれ？　6 桁しか表示されていませんね。もっと表示する桁数を増やしたいときには，cout を使う前に

```
cout.precision(表示桁数);
```

と指定します。この場合は 25 行目の上に追加して，例えば

```
cout.precision(20);
cout << "x = " << x << endl;
```

と書けば，20 桁表示になるはずです。ではやってみましょう。変更したら保存して再コンパイルするのを忘れないようにして下さい。これ以降は，特に必要がない限り，コンパイルのためのコマンドは省略します。

```
$ ./a.out
a = 3
b = -3.1
linear equation: 3 * x + -3.1 = 0
x = 1.033333333333333437
```

あれ？[2] 最後の方の桁がおかしいですね。ためしに 15 桁，30 桁, 50 桁表示にしてみると

```
15 桁: x = 1.03333333333333
20 桁: x = 1.033333333333333437
30 桁: x = 1.03333333333333343695414896501
50 桁: x = 1.03333333333333343695414896501461043953895568847 66
```

･･･ となって，結局 16 桁目までしか正しく計算できていないことが分かります。これはコンピュータが間違っているのではなく,「倍精度」浮動小数点数を用いている限り，16 桁以上の計算が出来ないことを意味しています。

16 桁以上の「精度」が欲しいときにはどうすればいいのか？　ということは後ほど解決先を示すことにして，当面は「**倍精度 (double) 浮動小数点数は 16 桁**」ということだけ覚えておきましょう。

5　2 次方程式を解く

では，1 次方程式同様，2 次方程式

$$ax^2 + bx + c = 0 \tag{2}$$

[2] 自分で書いてて，ベタなわざとらしさに失笑ものである。

を解くプログラムを作っていくことにしましょう。と言っても，C++プログラムに，例えば

$$x^2 - 2x - 3 = 0$$

という方程式を与えたら，自動的に左辺を因数分解して

$$(x - 3)(x - 1) = 0$$

とし，結果として $x = 3, 1$ を与えてくれる，なんてことをさせるのは至難の業です。ここでは 2 次方程式の解の公式

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

を計算するだけのプログラムを作ることにします。$\cdots$ といっても，どんな係数 a, b, c に対しても必ず二つの解を求めるってのは，結構面倒なお仕事だったりします。

ちなみに平方根 $\sqrt{a}$ の計算を C++で行うには，プログラムの先頭で

```
#include <cmath>
```

のように”cmath”ヘッダファイルをインクルードしておく必要があります。その上で

平方根 $\Longrightarrow$ `sqrt(a)`

と書かなくてはなりません。もうおわかりかと思いますが，こうした命令は全部英語から来ています。英語はコンピュータを使い倒すためにも必要不可欠なので，ちゃんと勉強しておきましょうね[3]。

5.1 実数解しかない場合

では，2 次方程式 (2) を解くプログラム”quadratic.cc”を作ります。当然ですが，3 つの係数 a, b, c を入れておく変数と，二つの解を入れておく変数 (**配列** (array) と呼びます)`sol[2]` を用意しておく必要があります。

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cmath>
 4:
 5: using namespace std;
 6:
 7: int main()
 8: {
```

[3]自分が出来なかったことを生徒に説教するという悪い癖が現れている。

```
 9:     double a, b, c;
10:     double d, sol[2];
11:
12:     cout << "a = ";
13:     cin >> a;
14:     cout << "b = ";
15:     cin >> b;
16:     cout << "c = ";
17:     cin >> c;
18:
19:     cout << "input equation: ";
20:     cout << a << " * x^2 + ";
21:     cout << b << " * x   + ";
22:     cout << c << " = 0 " << endl;
23:
24:     d = b * b - 4 * a * c;
25:
26:     sol[0] = (-b + sqrt(d)) / (2 * a);
27:     sol[1] = (-b - sqrt(d)) / (2 * a);
28:
29:     cout.precision(20);
30:     cout << "Solutions: " << sol[0] << ", " << sol[1] << endl;
31:
32:     return 0;
33: }
```

上記のプログラムでは，判別式 $d = b^2 - 4ac$ を計算して保存しておくための変数 d も用意してあります。

では先ほどの 2 次方程式

$$x^2 - 2x - 3 = 0$$

を解いてみましょう。

```
$ ./a.out
a = 1
b = -2
c = -3
input equation: 1 * x^2 + -2 * x   + -3 = 0
Solutions: 3, -1
```

出来ましたか？　では

$$x^2 + 123456x + 0.01 = 0$$

を解いてみて下さい。

```
$ ./a.out
a = 1
b = 123456
c = 0.01
input equation: 1 * x^2 + 123456 * x   + 0.01 = 0
Solutions: -8.10032e-08, -123456
```

ここで sol[0] に入っている値，”-8.10032e-08”ですが，これは指数表記と言って

$$-8.10032 \times 10^{-8}$$

を意味しています。とても絶対値の大きな (小さな) 実数を表記するときには短く書けて便利な表現方式です。

さてこの解ですが，実はそれほど正しくありません。もう少し表示桁数を大きく取って，20 桁表記にしてみると

```
$ ./a.out
a = 1
b = 123456
c = 0.01
input equation: 1 * x^2 + 123456 * x   + 0.01 = 0
Solutions: -8.1003236118704080582e-08, -123455.99999991900404
```

となりますが，やっぱり正しくないのです。実は

$$x_1 = -123455.99999991899948159662907 3645\cdots$$
$$x_2 = -8.10005184033709263541797468905874\cdots \times 10^{-8}$$

というのが本当の答えです。特に絶対値の小さい方の解がおかしくなってますね。上から 4 桁しか正しい値になっていません。これを「**桁落ち**」現象と呼びます。これを解決する手段はいくつかありますが，ここでは最後の節で，倍精度浮動小数点数よりずっと多い桁数で計算する「**多倍長浮動小数点数**」というものを使って強引に解決していきます。

さて，これとは別に，もっと別の問題もこの”quadratic.cc”には存在しています。判別式 $d = b^2 - 4ac$ がマイナスになる場合は全く対応できていません。もし

$$x^2 - 2x + 3 = 0$$

のような方程式を入力すると

```
$ ./a.out
a = 1
b = -2
c = 3
input equation: 1 * x^2 + -2 * x   + 3 = 0
Solutions: nan, nan
```

となって，”nan” = ”Not a number”，即ち，もうそれは数ではないもの，となってしまいます。これは

$$\sqrt{(-2)^2 - 4 \cdot 1 \cdot 3} = \sqrt{4 - 12} = \sqrt{-8}$$

を計算したために起こってしまっています。

桁落ち現象を解決する前に，まずこの問題を解決するための方法を見ていくことにしましょう。

5.2 複素数解になる場合

私たちが普段使っている実数 a は，ゼロでない限り正の数（プラス）と負の数（マイナス）のどちらかになり，必ず 2 乗すると正の数になります。即ち

$$a^2 > 0$$

となります。従って平方根を取ると

$$\sqrt{a^2} = |a|$$

となり，必ず正の数となります。もし負の数 c に対して

$$\sqrt{c}$$

が存在するとすれば

$$\left(\sqrt{c}\right)^2 = c < 0$$

となり，そんな $\sqrt{c}$ は実数ではあり得ない性質を持つことになります。そこで数学者はそのような，2 乗すると負の実数になる数を作ってしまいました。それを**虚数** (imaginary number) と呼びます。

先ほどの例にあった $\sqrt{-8}$ は

$$\sqrt{-8} = \sqrt{8} \cdot \sqrt{-1}$$

と分離できますので，実数 $\sqrt{8} = 2\sqrt{2}$ と，**虚数単位** $\sqrt{-1}$ の積として表現できます。

これを使うと先ほどの

$$x^2 - 2x + 3 = 0$$

の解 x は

$$x = \frac{2 \pm \sqrt{8} \cdot \sqrt{-1}}{2} = 1 \pm \sqrt{2} \cdot \sqrt{-1}$$

と書くことが出来ます。このように，虚数単位 $\sqrt{-1}$ を組み合わせて

$$\text{実数}\,1 + \text{実数}\,2 \cdot \sqrt{-1}$$

と表現する数を，**複素数** (complex number) と呼び，実数 1 を**実数部** (実部, real part), 実数 2 を**虚数部** (虚部, imaginary part) と呼びます。C++言語では，複素数を

(実数部, 虚数部)

と 2 次平面上の座標と同じように表現します。

複素数の計算は実数と同じように，実数部は実数部同士，虚数部は虚数部同士で計算し，掛け算の場合には

$$\left(\sqrt{-1}\right)^2 = -1$$

という虚数単位独特の性質を使って，多項式と同じように展開して計算します。C++言語ではこの計算の自動的に行ってくれますので，もし 2 次方程式 (2) の係数が複素数であれば，複素数の計算として扱ってくれます。

この C++の複素数計算機能を使って，先の”quadratic.cc”を，複素数係数 a, b, c 用のプログラム，”quadratic_complex.cc”に変更していきましょう。変更箇所は次の 3 点です。

1. 4 行目に

   ```
   #include <complex>
   ```

   を追加する。

2. 9 行目, 10 行目の

   ```
   double ...
   ```

   を

   ```
   complex<double> ...
   ```

   に変更する。

3. 25 行目, 27 行目，28 行目の整数定数 (”4”や”2”) を小数点表記 (”4.0”や”2.0”) にする。

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cmath>
 4: #include <complex>
 5:
 6: using namespace std;
 7:
 8: int main()
 9: {
10:     complex<double> a, b, c;
11:     complex<double> d, sol[2];
12:
13:     cout << "a = ";
14:     cin >> a;
15:     cout << "b = ";
16:     cin >> b;
17:     cout << "c = ";
18:     cin >> c;
19:
20:     cout << "input equation: ";
21:     cout << a << " * x^2 + ";
22:     cout << b << " * x   + ";
23:     cout << c << " = 0 " << endl;
24:
25:     d = b * b - 4.0 * a * c;
26:
27:     sol[0] = (-b + sqrt(d)) / (2.0 * a);
28:     sol[1] = (-b - sqrt(d)) / (2.0 * a);
29:
30:     cout << "Solutions: " << sol[0] << ", " << sol[1] << endl;
31:
32:     return 0;
33: }
```

では，先ほどの 2 次方程式を入力して，複素数の解を出力できるかどうか確認してみましょう。

```
$ g++ quadratic_complex.cc
$ ./a.out
a = 1
```

```
b = -2
c = 3
input equation: (1,0) * x^2 + (-2,0) * x   + (3,0) = 0
Solutions: (1,1.41421), (1,-1.41421)
```

ちゃんと $1+\sqrt{2}\cdot\sqrt{-1}$ と $1-\sqrt{2}\cdot\sqrt{-1}$ が求められていますか？　うまくいかなければもう一度変更事項を確認の上,

あなたが間違っている

ことを自覚してプログラムを見直して下さい。

応用問題

ここで作った”quadratic_complex.cc”は，$a=0$ となる場合には全く対応できていません。そこで $a=0$ の時は 1 次方程式

$$bx+c=0$$

の解,$x=-b/c$ を返し，$a=b=0$ の時は解が存在しないので，”Cannot solve!”を返すように改良を加えてたプログラム”quadratic__complex_all.cc”を作って下さい。

ヒント： 例えば判別式 d の計算をする前に

```
if(a == 0.0)
{
    if(b == 0.0)
    {
        cout << "Cannot solve!" << endl;
        return -1;
    }
    sol[0] = -c / b;

    cout << "Solution: " << sol[0] << endl;

    return 0;
}
```

という処理を付け加えるといいでしょう。

6　IEEE754 浮動小数点数と多倍長精度浮動小数点数

さて，複素数解になる場合の対処はできましたので，最後は桁落ち現象をチカラワザ[4]で解決するための方法を見ていくことにします。

倍精度浮動小数点数は 16 桁，ということを先ほど確認しました。従って，もっと長い桁を保存しておくためのデータ型が必要になります。ここでは MPFR/GMP という，**多倍長浮動小数点数** (multiple precision floating-point number) を実現してくれるソフトウェア (ライブラリ) を使うことにします。

論より証拠，まずは 1 次方程式を解くためのプログラム”linear.cc”を多倍長浮動小数点数 (`mpfr_class`) 向きに修正し，”linear_gmp.cc”を作成することにします。修正箇所は次の 3 点です。

1. 3 行目に

   ```
   #include "gmpfrxx.h"
   ```

   を挿入する。

2. 9 行目に

   ```
   mpfr_class::set_prec_dec(50);
   ```

   を入れ，10 行目の

   ```
   double a, b, x;
   ```

   を

   ```
   mpfr_class ga, gb, gx;
   ```

   に変更する[5]。

3. 文字列以外の a, b, x を ga, gb, gx に変更する。

最終的には次のようなプログラムになります。

[4]多倍長計算なんてまやかしだ！と言う方はここで退散して下さい。そんな偏狭な奴は私は嫌いだ。

[5]変数名を変える必要はないが，後で倍精度プログラムと合体させるときに面倒にならないよう配慮した。

```
 1: #include <iostream>
 2: #include <cmath>
 3: #include "gmpfrxx.h"
 4:
 5: using namespace std;
 6:
 7: int main()
 8: {
 9:     mpfr_class::set_dprec_dec(50);
10:     mpfr_class ga, gb, gx;
11:
12:     cout << "a = ";
13:     cin >> ga;
14:     cout << "b = ";
15:     cin >> gb;
16:
17:     cout << "linear equation: " << ga << " * x + " << gb << " = 0"
 << endl;
18:
19:     if(ga == 0.0)
20:     {
21:         cout << "Cannot solve!" << endl;
22:         return -1;
23:     }
24:
25:     gx = -gb / ga;
26:
27:     cout.precision(50);
28:     cout << "x = " << gx << endl;
29:
30:     return 0;
31: }
```

これをコンパイルする時には

```
$ g++ linear_gmp.cc -lgmpfrxx -lgmpxx -lmpfr -lgmp
```

もしくは

```
$ g++ linear_gmp.cc -L/usr/local/lib -lgmpfrxx -lgmpxx -lmpfr -lgmp
```

と指定して下さい。ここでは 4 つのライブラリ (gmpfrxx[4], gmpxx[2], MPFR[3], GMP[2]) を使って多倍長精度浮動小数点数の計算を実現しているので，コンパイル時の指定は複雑になります。ご辛抱下さい。

これを実行すると

```
$ ./a.out
a = 3
b = -3.1
linear equation: 3.00000 * x + -3.10000 = 0
x = 1.03333
```

となります。一見何も変わっていないようですが，9 行目の指定

```
mpfr_class::set_prec_dec(50);
```

によって，ga, gb, gx は 50 桁の小数 (浮動小数点数) を保持できるようになっていて，倍精度浮動小数点数に比べて精度はぐんと増しています。その証拠に，出力 (27 行目) を 50 桁表示してみると (どうやればいいか，覚えてますか？),

```
$ ./a.out
a = 3
b = -3.1
linear equation: 3.00000 * x + -3.10000 = 0
x = 1.03333333333333333333333333333333333333333333333333
```

となって，確かに 50 桁ぴったり合っていることが分かります[6]。

ではこの「ぴったり合っている桁数」を数値で表現してみましょう。そのためには**相対誤差** (relative error) というものを使います。

相対誤差とは，中途半端に正しい数値 (**近似値**) と，近似値よりは真の値に近い近似値 (これを**真値**と仮定します) を使って

$$\text{相対誤差} = \left| \frac{\text{近似値} - \text{真値}}{\text{真値}} \right| \tag{4}$$

として計算した値のことを言います。ちなみに真値がゼロの場合は分子の絶対値 (近似値と真値の差) を相対誤差として採用することにします。

私の方で (4) 式に基づいて相対誤差を計算する関数 relative_error を用意しましたので，それを使って倍精度浮動小数点数で計算した解の相対誤差を求めてみましょう。

”linear_gmp.cc”に，次の変更を行って下さい。

1. 4 行目に

[6] 10 進桁を指定する機能はオリジナルの gmpfrxx には備わっていないので注意。

```
#include "relative_error.h"
```

を挿入する。これで相対誤差計算のための関数 relative_error が使えるようになる[7]。

2. 8 行目と 9 行目の間に

```
double a, b, x;
```

を挿入し，倍精度浮動小数点数で計算するための変数を復活させる。

3. 16 行目に次の 2 行を挿入する。

```
a = ga.get_d();
b = gb.get_d();
```

これで，多倍長精度浮動小数点数で入力した方程式の係数を，倍精度浮動小数点数に短く丸めて格納できる。

4. 24 行目に

```
x = -b / a;
```

を挿入して，倍精度浮動小数点数で計算した解を求める。そして 27 行目と 28 行目の間にこれを出力するため,

```
cout << "x = " << x << endl;
```

を挿入する。

5. 相対誤差を計算し，短い桁数 (5 桁) で出力するため，次の行を 29 行目に追加。

```
cout.precision(5);
cout << "Relative Error: " << relative_error(x, gx) << endl;
```

相対誤差を計算する関数 relarive_error は必ず

relative_error(近似値，真値)

という形式で使って下さい。

以上の改良を加えた”linear_gmp.cc”をコンパイルして実行すると次のように，倍精度浮動小数点数で計算した解と，50 桁の多倍長浮動小数点数で計算した解，そして倍精度浮動小数点数の解に含まれる相対誤差が出力されます。

[7]関数定義を教えるのが本義ではないのでこのようにした。プログラムが好きな生徒相手ならここを自力で考えさせるのもいいだろう。

```
$ ./a.out
a = 3
b = -3.1
linear equation: 3.00000 * x + -3.10000 = 0
x = -1.033333333333333343695414896501461043953895568847 66
x = -1.033333333333333333333333333333333333333333333333 33
Relative Error: 1.0028e-16
```

この場合，相対誤差は

$$1.0028 \times 10^{-16}$$

であることが分かります。ぴったり合っている桁数を知りたいときは，$\log_{10}$(相対誤差)の整数部分を採用して下さい。

練習問題2

以上見てきたように，多倍長浮動小数点数を用いることで，正しい計算桁数が維持できることが確認できました。そこで，2次方程式の実数解を求めるプログラム”quadratic.cc”を多倍長浮動小数点数で計算できるようにし，倍精度浮動小数点数で計算した値の相対誤差を出力するプログラム”quadratic_gmp.cc”を作成して下さい。

この”quadratic_gmp.cc”を用いると，

$$x^2 + 123456x + 0.01 = 0$$

の倍精度浮動小数点数の解は

```
$ ./a.out
a = 1
b = 123456
c = 0.01
input equation: 1 * x^2 + 123456 * x   + 0.01 = 0
Solutions: -8.10032e-08, -123456
Solutions: -8.1000518403370926354179746890587455661643741983478e-8, -1.23
455999999918999481596629073645820253109412544 34e5
Relative Errors in sol[0]: 3.3552e-5
Relative Errors in sol[1]: 3.6922e-17
```

となって，5桁程度しか出ていないことが，相対誤差の大きさでもはっきり分かります。

7　複素数を係数とする 2 次方程式の場合

では，最後に 2 次方程式の複素数解まで対応したプログラムを多倍長浮動小数点数で計算できるようにした”quadratic_complex_gmp.cc”を作成し，倍精度浮動小数点数計算で生じた相対誤差を出力するようにしてみましょう。改良のポイントは次の通りです。

1. 多倍長浮動小数点数の複素数で平方根を精度良く計算するために私の方で用意したヘッダファイル complex_gmpfrxx.h をインクルードする必要がある。

2. 複素数の実数部は

   複素数変数名.real()

   ，虚数部は

   複素数変数名.imag()

   と呼び出すこと。多倍長精度浮動小数点数から倍精度浮動小数点変数への変換，相対誤差の計算は，実数部，虚数部それぞれで実施すること。

3. 多倍長精度浮動小数点数の複素数計算では定数を複素数に変更して行う必要がある。ここでは 40 行目から 42 行目の

   complex<mpfr_class>(4UL, 0UL) や complex<mpfr_class>(2UL, 0UL)

   という記述がそれに当たる。

```
 1: #include <iostream>
 2: #include <iomanip>
 3: #include <cmath>
 4: #include <complex>
 5: #include "gmpfrxx.h"
 6: #include "relative_error.h"
 7: #include "complex_gmpfrxx.h"
 8:
 9: using namespace std;
10:
11: int main()
12: {
13:     complex<double> a, b, c;
14:     complex<double> d, sol[2];
```

```
15:
16:     mpfr_class::set_dprec_dec(50);
17:     complex<mpfr_class> ga, gb, gc;
18:     complex<mpfr_class> gd, gsol[2];
19:
20:     cout << "a = ";
21:     cin >> ga;
22:     cout << "b = ";
23:     cin >> gb;
24:     cout << "c = ";
25:     cin >> gc;
26:
27:     a.real() = ga.real().get_d(); a.imag() = ga.imag().get_d();
28:     b.real() = gb.real().get_d(); b.imag() = gb.imag().get_d();
29:     c.real() = gc.real().get_d(); c.imag() = gc.imag().get_d();
30:
31:     cout << "input equation: ";
32:     cout << a << " * x^2 + ";
33:     cout << b << " * x   + ";
34:     cout << c << " = 0 " << endl;
35:
36:     d = b * b - 4.0 * a * c;
37:     sol[0] = (-b + sqrt(d)) / (2.0 * a);
38:     sol[1] = (-b - sqrt(d)) / (2.0 * a);
39:
40:     gd = gb * gb - complex<mpfr_class>(4UL, 0UL) * ga * gc;
41:     gsol[0] = (-gb + sqrt(gd)) / (complex<mpfr_class>(2UL, 0UL) * ga);
42:     gsol[1] = (-gb - sqrt(gd)) / (complex<mpfr_class>(2UL, 0UL) * ga);
43:
44:     cout.precision(50);
45:     cout << "Solutions: " << sol[0] << ", " << sol[1] << endl;
46:     cout << "Solutions: " << gsol[0] << ", " << gsol[1] << endl;
47:
48:     cout.precision(5);
49:     cout << "Relative Errors in sol[0]: " << relative_error(sol[0].
real(), gsol[0].real()) << ", " << relative_error(sol[0].imag(), gsol[0].
imag()) << endl;
50:     cout << "Relative Errors in sol[1]: " << relative_error(sol[1].
real(), gsol[1].real()) << ", " << relative_error(sol[1].imag(), gsol[1].
```

```
imag()) << endl;
51:
52:     return 0;
53: }
```

このプログラムを使って先の例題を解いてみましょう。

```
$ ./a.out
a = 1
b = -2
c = 3
input equation: (1,0) * x^2 + (-2,0) * x   + (3,0) = 0
Solutions: (1,1.4142135623730951454746218587388284504413604736328),
(1,-1.4142135623730951454746218587388284504413604736328)
Solutions: (1.0000000000000000000000000000000000000000000000000000,1.41
42135623730950488016887242096980785696718753770), (1.000000000000000
0000000000000000000000000000000000000,-1.414213562373095048801688724209
6980785696718753770)
Relative Errors in sol[0]: 0, 6.8358e-17
Relative Errors in sol[1]: 0, 6.8358e-17
```

これで一通りの機能は揃いました。では，最後の数値実験を行って結果をまとめて下さい。

8　数値実験

では，次の 1 次方程式，2 次方程式を，今まで作ってきた C++プログラムを動かして解いてみましょう。その際には，求められた解と相対誤差を記録しておいて下さい。

1. $3x - 5 = 0$ ($x = 5/3 = 1.66666\cdots$)
2. $2x + 3 = 0$
3. $x^2 + 3x + 4 = 0$
4. $x^2 - 1.11x + 0.101 = 0$
5. $x^2 - 3x + 2.26 = 0$ ($x_1 = 1.5 + 0.1\sqrt{-1},\ x_2 = 1.5 - 0.1\sqrt{-1}$)

参考文献

[1] Gnu compiler collection. http://gcc.gnu.org/.

[2] Swox AB. GNU MP. http://gmplib.org/.

[3] MPFR Project. The MPFR library. http://www.mpfr.org/.

[4] Jon Wilkening. gmpfrxx. http://math.berkeley.edu/~wilken/code/gmpfrxx/.

A GNU MP, MPFR を使用する環境作り

本資料では，Linux 環境で GCC を使って実習を行うことを想定している。そのためあらかじめ GMP[2], MPFR[3] はインストールしておく必要がある。インストール方法は URL にあるドキュメントを参照のこと。

B 特別に用意したヘッダファイルとパッチ

資料中にあるプログラムの機能を実現するために使ったヘッダファイルとパッチを示す。

B.1 relative_error.h

```
 1: //#include "complex_gmpfrxx.h"
 2: #include <complex>
 3: #include <cmath>
 4: #include "gmpfrxx.h"
 5:
 6: mpfr_class relative_error(mpfr_class approx, mpfr_class true_sol)
 7: {
 8:     mpfr_class ret(approx.get_prec());
 9:
10:     ret = approx - true_sol;
11:     if(true_sol != 0)
12:         ret = abs(ret / true_sol);
13:
14:     return ret;
15: }
```

B.2 complex_gmpfrxx.h

```
 1: //#include "sqrt_gmpfrxx.h"
 2: #include <complex>
 3: #include <cmath>
 4: #include "gmpfrxx.h"
 5:
 6: std::complex<mpfr_class> sqrt(std::complex <mpfr_class> __z)
 7: //complex sqrt(const complex __z)
 8: {
 9:     unsigned long prec;
10:
11:     mpfr_class __x, __y, __t, __u;
12:
13:     prec = __z.real().get_prec();
14:     if(prec < __z.imag().get_prec())
15:         prec = __z.imag().get_prec();
16:
17:     __x.set_prec(prec);
18:     __y.set_prec(prec);
19:     __t.set_prec(prec);
20:     __u.set_prec(prec);
21:
22:     __x = __z.real();
23:     __y = __z.imag();
24:
25:     if (__x == 0UL)
26:     {
27:         __t = sqrt(abs(__y) / 2UL);
28:         if(__y < 0UL)
29:             return std::complex<mpfr_class>(__t, -__t);
30:         else
31:             return std::complex<mpfr_class>(__t, __t);
32:     }
33:     else
34:     {
35:         __t = sqrt(2UL * (std::abs(__z) + abs(__x)));
36:         __u = __t / 2UL;
37:         if(__x > 0UL)
```

```
38:             return std::complex<mpfr_class>(__u, __y / __t);
39:         else
40:         {
41:             if(__y < 0UL)
42:                 return std::complex<mpfr_class>(abs(__y) / __t,  -__u);
43:             else
44:                 return std::complex<mpfr_class>(abs(__y) / __t,  __u);
45:         }
46:     }
47: };
48:
```

B.3 gmpfrxx.h に当てたパッチ

```
 1: 60,62d59
 2: < // appended by T.Kouya
 3: < #include <cmath>
 4: <
 5: 82,86d78
 6: <
 7: <   // appended by T.Kouya
 8: <   static mpfr_prec_t  get_dprec_dec() { return (mpfr_prec_t)((do
uble)mpfr_get_default_prec() * log10(2.0)); }
 9: <   static void         set_dprec_dec(mpfr_prec_t p_dec=16)
10: <   { mpfr_set_default_prec((unsigned long)ceil((double)p_dec / lo
g10(2.0))); }
11: 2197,2199d2188
12: < // apnended by T.Kouya
13: <   unsigned long int get_prec_dec() const { return (unsigned long)
((double)mpfr_get_prec(mp) * log10(2.0)); }
14: <
15: 2202,2208d2190
16: <
17: < // apnended by T.Kouya
18: < //  void set_prec_dec(unsigned long prec_dec) { mpfr_set_prec(mp,
(unsigned long)ceil((double)prec_dec / log10(2.0))); }
19: <   void set_prec_dec(int prec_dec) { mpfr_set_prec(mp, (unsigned lon
g)ceil((double)prec_dec / log10(2.0))); }
20: <   void set_prec_dec(unsigned int prec_dec) { mpfr_set_prec(mp, (uns
```

```
igned int)ceil((double)prec_dec / log10(2.0))); }
21: <   void set_prec_dec(long unsigned prec_dec) { mpfr_set_prec(mp, (lo
ng unsigned int)ceil((double)prec_dec / log10(2.0))); }
22: <
23: 2214,2215c2196
24: < //  __gmp_expr() { mpfr_init(mp); mpfr_set_d(mp, 0.0, MpFrC::get_rn
d()); }
25: <   __gmp_expr() { mpfr_init(mp); mpfr_set_ui(mp, 0UL, MpFrC::get_rnd
()); }
26: ---
27: >   __gmp_expr() { mpfr_init(mp); mpfr_set_d(mp, 0.0, MpFrC::get_rnd(
)); }
28: 4337,4341d4317
29: <
30: < /*********************************************/
31: < /* Appended by T.Kouya                       */
32: < /*********************************************/
33: <
```