C, C++ で複素数

桂田 祐史

2010年9月7日, 2018 年 6 月 19 日, 2024年9月15日

https://m-katsurada.sakura.ne.jp/labo/text/complex-c.pdf

目 次

C 言語に複素数が導入されたのは、(C の結構長い歴史からすれば) 比較的最近のことで使い方も浸透していないし、さらにそれが省いても良いことになったりして (まあ、GCC や LLVM では使い続けられそうな気がするけれど)、率直に言って C 言語で複素数を使うのはあまりお勧めではない。

でも、これまで学生が僕のところに持ち込んで来るプログラムは大抵 C だったりする (笑)。まあ、C++習っていないから仕方ないか。

ベクトル・行列とか複素数とか、色々考えると、C よりは C++とか、Fortran とか、MATLAB とかを使う方が良い気がする。そうそう、Python や Julia でも良いだろうし。

1 参考にした情報源

久しぶりに見たら、今でも残っていて感心 ((2024/6/15 追記) さらに 6 年経っても残っている…これは多分ずっと残るな)。

- **プログラミング言語 C の新機能**[1]
- **Status of C99 features in GCC**[2]

[1] http://seclan.dll.jp/c99d/
[2] http://gcc.gnu.org/c99status.html

- **C approved standards** (承認された規格)[3]

もっとも、C11 では、複素数の機能は省いて良い機能になったそうだ。GCC や LLVM ではサポートが続くようだけど…

2 C で複素数

導入されたのが、C++よりも後だったので、「C++は C の上位互換」というのが崩れてしまったのが悲しい。

2.1 complex.h を使わずにやる場合

変数の宣言は、次のように行う。

```
float _Complex a;
double _Complex b;
long double _Complex c;
```

変数 a の実部、虚部はそれぞれ__real__ a, __imag__ a で得られる。

```
/*
 * test-without-complex.h.c
 *    __real__, __imag__ 演算子
 *    符号を必ず表示させる %+f という書式
 */

#include <stdio.h>

int main(void)
{
  double _Complex a,b,c;

  a = 1+2i;

  printf("input Re b and Im b:");
  scanf("%lf%lf", &__real__ b, &__imag__ b);

  c = a + b;
  printf("%f%+fi\n", __real__ c, __imag__ c);
  return 0;
}
```

GCC には、虚数単位を表す _Complex_I がないが (直るの？直ったの？)、

```
#define _Complex_I (1.0iF)
```

のように自分で定義すれば良い。

2.2 complex.h を使ってやる場合

変数の宣言は、次のように行う。

[3] http://www.open-std.org/JTC1/SC22/WG14/www/standards

```
float complex a;
double complex b;
long double complex c;
```

実部、虚部を求める関数として、

単精度	crealf(), cimagf()
倍精度	creal(), cimag()
長倍精度	creall(), cimagl()

がある。その他に絶対値 cabs(), 共役複素数 conj(), 偏角 carg(), それからもちろん (?) cexp(), csin() などの関数が用意されている。

(システム標準のコンパイラーならば、complex.h のパス名は/usr/include/complex.h であろうから、一度どういうものがあるか読んでみると良い。)

虚数単位は I という名前で使える。i は他で使われるだろうから大文字ということらしい (でも 1i とすれば虚数単位になるのかな)。もし j が使いたければ

```
#undef I
#define j _Imaginary_I
```

としなさい、だとか。

```
/*
 * test-with-complex.h.c
 */

#include <stdio.h>
#include <complex.h>

int main(void)
{
  double complex a,b,c;
  double re_b, im_b;

  a = 1+2i;

  printf("input Re b and Im b:");
  scanf("%lf%lf", &re_b, &im_b);
  b = re_b + im_b * I;

  c = a + b;
  printf("%f%+fi\n", creal(c), cimag(c));
  return 0;
}
```

(某月某日) 某学生がはまった落とし穴。complex double 型の数値を double 型の数値に代入して、結果がおかしいと悩んでいた (代入の結果は、複素数値の実部になる。代入直後に結果を表示したりすれば気がついただろうが、計算の途中に紛れ込んでいたため、発見が遅れた。)。これくらいはコンパイラーが警告して欲しいような気がするが、もともと

```
int i;
double x;
...
i = x;
```

のように int 型変数に double 型の数値を代入できたりする (結果は原点に向かっての切り捨てに

なった — いつもこうなるかどうかは知らない) 言語仕様なので、そう思うのは筋違いなのかもしれない。注意しないと。

3 C++

std::complex というテンプレート・ライブラリィがある。

```
#include <complex>
using namespace std;   // これはお好み
```

単精度	complex<float>
倍精度	complex<double>
?精度	complex<long double>

C の a = 1+2i; は、C++ では

```
 a = complex<double>(1,2);
```

で実現出来る。変数定義と同時に値の設定をしたいのならば、

```
 complex<double> a(1,2);
```

とするのも良い。

虚数単位を使いたい場合は、自分で定義するのかな？complex<double> I(0,1); とか。

test-cpp-complex-1.cpp

```
/*
 * test-cpp-complex-1.cpp
 *   g++ test-cpp-complex-1.cpp
 */

#include <iostream>
#include <iomanip> // setprecision()
#include <complex>
#include <cmath>

using namespace std;

int main(void)
{
  complex<double> I(0,1), z;
  cout << "i=" << I << endl;
  cout << "i^2=" << I * I << endl;
  cout << setprecision(16) << "sqrt(I)=" << sqrt(I) << endl;
  z=1.0+I;
  cout << "z=" << z << ", z^2=" << z*z << endl;
  return 0;
}
```

コンパイル&実行例

```
% g++ test-cpp-complex-1.cpp
% ./a.out
i=(0,1)
i^2=(-1,0)
sqrt(I)=(0.7071067811865476,0.7071067811865475)
z=(1,1), z^2=(0,2)
%
```

確かに

$$\texttt{sqrt(I)} = \frac{1+i}{\sqrt{2}}, \quad (1+i)^2 = 2i$$

と計算出来ている。

実部、虚部、複素数、共役複素数の例もつけておく。

test-cpp-complex-3.cpp

```
// test-cpp-complex-3.cpp

#include <iostream>
#include <complex>

using namespace std;

int main(void)
{
  complex<double> z(1.0, 2.0);
  cout << "z=" << z << endl;
  cout << "z.real()=" << z.real() << ", real(z)=" << real(z) << endl;
  cout << "z.imag()=" << z.imag() << ", imag(z)=" << imag(z) << endl;
  cout << "abs(z)="  << abs(z)  << endl;
  cout << "conj(z)=" << conj(z) << endl;
}
```

A C++プログラム例

A.1 古いC++プログラムの修正

昔作った C++ プログラム[4]を引っ張りだして来たら、コンパイルできない (苦笑)。C++ ってそういうプログラミング言語だと、2000 年頃痛感したけれど、その頃作ったプログラムも今はコンパイルできない。C ではあり得ないことだけど、C++ では、それが許される環境にあるので、新しい機能が取り込める、ということなのかな？

- 以前は

  ```
  #include <iostream.h>
  #include <complex.h>
  ```

  のようにしていたものを

[4]「非線形方程式、特に代数方程式の数値解法」, http://nalab.mind.meiji.ac.jp/~mk/lecture/ouyoukaiseki4/notebook/nonlinear-algebra.pdf の中で代数方程式を解くプログラムを C++ で書いた。

```
#include <iostream>
#include <complex>
```

と直す。

- 以前は std という namespace は指定する必要がなかったが、さぼれなくなった。std::cin のように一々 std:: をつけるか、

```
using namespace std;
```

とする。

- 式の中に complex と int を混在させると、必要に応じて int から complex に型上げしてくれていたが、それをしてくれなくなった。自分で型を直すのであれば、2 は、(double) 2 や double(2) のようにキャストするか、2.0 とするか、あるいは complex<double>(2,0) のようにする。

  自動的に処理することも出来て、例えば Michel and Stoitsov [1] の **hyp_2F1** の complex_functions.H では、それにオペレーター・オーバーロードで対応している。

complex_functions.H から引用

```
// Usual operator overloads of complex numbers with integers
// ----------------------------------------------------------
// Recent complex libraries do not accept for example z+n or z==n with n integer, signed or unsign
// The operator overload is done here, by simply putting a cast on double to the integer.
```

例えば

```
inline complex<double> operator + (const complex<double> &z,const int n)
{
  return (z+static_cast<double> (n));
}
```

のようなことをやっている (この手の関数定義を約 30 個)。

- C 由来の exit() を使うには #include <cstdlib> が必要。

- "DKGRAPH" のような文字列を表す式が、C++ では string const 型となっている。それを C で書いた char へのポインター型の引数を想定している関数に渡すと、コンパイル・エラーにはならないが、警告が出る。gcc では -Wno-write-strings で強制的に黙らせることが出来るけれど、プログラムの直し方を後述の DK3.C を例に説明する。直し方は二つある。(1) C プログラムの宣言の方を

```
extern void     g_init G_ARGS ((char * file_name, G_P_DIM win_width,
                                G_P_DIM win_height));
```

から

```
extern void     g_init G_ARGS ((const char * file_name, G_P_DIM win_width,
                                G_P_DIM win_height));
```

のように const をつける。 (2) 呼び出す側で

```
g_init("DKGRAPH", 120.0, 120.0);
```

に

```
g_init((char *)"DKGRAPH", 120.0, 120.0);
```

のように (const 抜きの) char * へのキャストをつける。(2) はやっつけ仕事で、(1) の方が望ましいのかな、でも自分が作ったのではないライブラリィを改変するのは副作用の発生が怖い (この例は GLSC のヘッダーファイル glsc.h を直して良いか、ということになるので悩ましい)。

A.2 quadratic-equation.C

```
// quadratic-equation.C --- 複素係数の 2 次方程式を解く。
//
//      g++ -o quadratic-equation quadratic-equation.C
//
//    注意: 素朴な（桁落ちの対策などしていない）アルゴリズムを使っている。
//
//    2016/3/22 修正
//      2000 年頃動いていたプログラムがコンパイル出来なかったので修正。
//      iostream.h, complex.h をインクルードしていた → .h を取る
//      namespace を指定していなかった（必要なかった）→ using namespace std;
//      4, 2 などの int 型を使っていた → double 型にする
//      最近は拡張子が大文字の C なのは珍しい気がする（.cpp とかが多い？）けど。

#include <iostream>
#include <complex>
using namespace std;

int main(void)
{
  complex<double> a, b, c, D, x1, x2;

  cout << "複素係数の 2 次方程式 a x^2+b x+c=0 (a ≠ 0) を解きます。" << endl;
  cout << " 複素数は ( ) でくくり、実部と虚部をカンマ , で区切って表す。"
       << endl;
  cout << " 例えば 1+2i は (1,2) と表わします。" << endl;
  cout << "入力してください。" << endl;

  cout << "a="; cin >> a;
  cout << "b="; cin >> b;
  cout << "c="; cin >> c;

  cout << "a=" << a << ", b=" << b << ", c=" << c << endl;

  D = sqrt(b * b - 4.0 * a * c);
  x1 = (-b + D) / (2.0 * a);
  x2 = (-b - D) / (2.0 * a);
  cout << "x1=" << x1 << endl;
  cout << "x2=" << x2 << endl;

  return 0;
}
```

A.3 complex-newton.C

```
/*
 * complex-newton.C -- Newton 法で方程式 f(x)=0 を解く
 *  コンパイル: g++ -o complex-newton complex-newton.C
 *  実行: ./complex-newton
 *
 *  2016/3/22 修正
 */

#include <iostream>
#include <complex>
using namespace std;

int main(void)
{
    int i, maxitr = 100;
    complex<double> f(complex<double>), dfdz(complex<double>), x, dx;
    double eps;

    cout << " 初期値 x0, 許容精度 ε =";
    cin >> x >> eps;

    cout.precision(16);
    for (i = 0; i < maxitr; i++) {
        dx = - f(x) / dfdz(x);
        x += dx;
        cout << "f(" << x << ")=" << f(x) << endl;
        if (abs(dx) <= eps) break;
    }
    return 0;
}

/*
 * この関数の例は杉原・室田『数値計算法の数理』岩波書店, p.67 による
 *   f(z) = z^3-2z+2
 *   1 実根 (≒-1.769292354238631),2 虚根 (≒ 0.8846461771193157 ± 0.5897428050222054)
 *   を持つが、原点の近くに初期値を取ると、0 と 1 の間を振動する。
 */

complex<double> f(complex<double> z)
{
  /* return z * z * z - 2 * z + 2; */
  return z * (z * z - 2.0) + 2.0;
}

/* 関数 f の導関数 (df/dx のつもりで名前をつけた) */
complex<double> dfdz(complex<double> z)
{
  return 3.0 * z * z - complex<double>(2,0);
}
```

コンパイル&実行例

```
% g++ complex-newton.C
% ./a.out
 初期値 x0, 許容精度ε=0.3 1e-14
f((1.12485549132948,0))=(1.173568531458942,0)
f((0.4713843772268934,0))=(1.161974377253067,0)
f((1.342827923349606,0))=(1.735713781960667,0)
f((0.8337553363070073,0))=(0.9120726492429374,0)
f((-9.840767203756229,0))=(-931.3052418605159,0)
f((-6.612920012816673,0))=(-273.9618545552224,0)
f((-4.4923431047393,0))=(-79.67594843313302,0)
f((-3.131371682066463,0))=(-22.44188600195348,0)
f((-2.312816652470607,0))=(-5.745902514520807,0)
f((-1.903778836863019,0))=(-1.092448577226583,0)
f((-1.780659980946024,0))=(-0.0847076152532642,0)
f((-1.769384049454865,0))=(-0.0006777810558658004,0)
f((-1.769292360276141,0))=(-4.462436198338082e-08,0)
f((-1.769292354238631,0))=(2.220446049250313e-16,0)
f((-1.769292354238631,0))=(2.220446049250313e-16,0)
mk@katsuradanoMacBook-Pro-555 progs-algebraic-equation % ./a.out
 初期値 x0, 許容精度ε=0.1 1e-14
f((1.014213197969543,0))=(1.014822114238246,0)
f((0.07965576631987636,0))=(1.841193886472037,0)
f((1.009098740372765,0))=(1.009347854859992,0)
f((0.05222652653371329,0))=(1.895689400532465,0)
f((1.003965184727484,0))=(1.004012415140623,0)
f((0.02332943565497392,0))=(1.953353826028611,0)
f((1.000804353182403,0))=(1.000806294654933,0)
f((0.004806794546775128,0))=(1.990386521968754,0)
f((1.000034548045781,0))=(1.000034551626525,0)
f((0.0002072524744252124,0))=(1.999585495060052,0)
f((1.000000064421484,0))=(1.000000064421497,0)
f((3.865287804272199e-07,0))=(1.999999226942439,0)
f((1.00000000000224,0))=(1.00000000000224,0)
f((1.34559030584569e-12,0))=(1.999999999997309,0)
f((1,0))=(1,0)
f((0,0))=(2,0)
f((1,0))=(1,0)
f((0,0))=(2,0)

中略

f((0,0))=(2,0)
f((1,0))=(1,0)
f((0,0))=(2,0)
%
```

初期値 0.3 で始めると実根に収束するが、初期値 0.1 で始めると、途中から $(0,0)$ と $(1,0)$ の間を振動する。確かに本に書いてある通り。

A.4 DK3.C

(拙いところがあって、今見ると直したいのだけど…あくまでも C++ で複素数を扱うにはどうするかのサンプルと考えれば、まあいいかと。)

代数方程式を連立法 (Durand-Kerner 法ともいう) で解くプログラムである。

「非線型方程式、特に代数方程式の解法」[5] の付録 B「応用解析 IV の数値実験」に解説がある。実行結果は https://m-katsurada.sakura.ne.jp/labo/text/nonlinear-algebraic/node85.html

[5] https://m-katsurada.sakura.ne.jp/labo/text/nonlinear-algebraic/

に載っている。

GLSC と X11 を使っているので、コンパイルするときに、インクルード・ファイルやライブラリィの置き場所を指定する必要がある (簡単には実行して試せないです。あしからず。)。最近の私の環境 (Mac で XQuartz というのを使っていて、GLSC を /usr/local/include と /usr/local/lib に置いている) では

```
g++ -I/usr/local/include -I/usr/X11/include DK3.C -L/usr/local/lib -L/usr/X11/lib -lglscd -lX11
```

のようにしてコンパイルする。

```
// DK3.C -- DKA 法で代数方程式の根を求める
//
//   数学科 6701 号室のワークステーション環境でのコンパイル
//     (グラフィックス・ライブラリィ GLSC を使っているので
//      情報科学センターではコンパイルできない。悪しからず。)
//     g++ -o DK3 DK3.C -I/usr/local/include -lglscd -lX11 -lsocket
//   あるいは
//     ccmg+ DK3.C
//
//  Win32 環境下の glscwin でも (若干の修正の下に) 動作させられることが
//  分かったので、画面に表示するメッセージを変更しました (2000/10/18)。
//
//  2016/3/22 修正
//  (1) #include するのは .h なし
//  (2) using namespace std;
//  (3) #include <cstdlib> // exit() のため
//  (4) int 型の n を (double)n とキャスト
//  (5) "DKGRAPH" を (char *)"DKGRAPH" (string は char * と違うのだそうな)
//  g++ -I /usr/X11/include DK3.C -L/usr/local/lib -lglscd -L/usr/X11/lib -lX11
//
// 2016/12/30 修正
// extern { ... }; 最後の ; を忘れていたのを修正。


#include <iostream>
#include <iomanip>   // setprecision() のため
//#include <math>
#include <complex>   // complex<double> のため
#include <cstdlib>   // std::exit() のため
using namespace std;

// GLSC のヘッダーファイルをインクルード
extern "C" {
#define G_DOUBLE
#include <glsc.h>
};

// プロトタイプ宣言
complex<double> polynomial(int, complex<double> *, complex<double>);
complex<double> bunbo(int, complex<double> *, int);

// 解きたい代数方程式の次数の最高値
#define MAXN 100

int main(void)
{
  int i, k, n;
  complex<double> a[MAXN+1], x[MAXN], newx[MAXN], dx[MAXN];
  complex<double> g, I(0,1);
  double r0, R, max, pi;
```

```
  // 数学定数（円周率）の準備
  pi = 4 * atan(1.0);

  // 表\示の桁数を 16 桁にする
  cout << setprecision(16);

  // 方程式の入力
  cout << "次数 n を入力してください（1 ≦ n ≦" << MAXN << "): ";
  cin >> n;
  if (n > MAXN || n <= 0) {
    cerr << "次数は" << MAXN << "以下の自然数として下さい。" << endl;
    exit(0);
  }
  for (i = 0; i <= n; i++) {
    cout << (n-i) << "次の係数を入力してください: ";
    cin >> a[i];
  }

  // 多項式を最高次の係数で割って monic にする
  cout << "monic にします。" << endl;
  for (i = 1; i <= n; i++)
    a[i] /= a[0];
  a[0] = 1;
  cout << "修正した係数" << endl;
  for (i = 0; i <= n; i++)
    cout << "a[" << i << "]=" << a[i] << endl;

  // Aberth の初期値を配置する円の決定
  g = - a[1] / (double)n;
  cout << "根の重心" << g << endl;
  max = 0;
  for (i = 1; i <= n; i++)
    if (abs(a[i]) > max)
      max = abs(a[i]);
  cout << "max|a_i|=" << max << endl;
  r0 = abs(g) + 1 + max;
  cout << "根は重心中心で、半径 r0=" << r0 << "の円盤内にある" << endl;

  cout << "円の半径（分からなければ上の値を指定してください): ";
  R = r0;
  cin >> r0;
  if (r0 > R)
    R = r0;
  cout << "図は根の重心を中心として半径 " << R
#ifdef __CYGWIN__
       << "の円が表\示できるようにします。" << endl;
#else
       << "の円が表示できるようにします。" << endl;
#endif

  // Aberth の初期値
  cout << "初期値" << endl;
  for (i = 0; i < n; i++) {
    double theta;
    theta = 2 * i * pi / n + pi / (2 * n);
    x[i] = g + r0 * exp(I * theta);
    cout << x[i] << endl;
  }

  // グラフィックス・ライブラリィ GLSC の初期化
  g_init((char *)"DKGRAPH", 120.0, 120.0);
  g_device(G_BOTH);
  // 座標系の指定
```

```
  g_def_scale(0,
       real(g) - 1.1 * R, real(g) + 1.1 * R,
       imag(g) - 1.1 * R, imag(g) + 1.1 * R,
       10.0, 10.0, 100.0, 100.0);
  g_sel_scale(0);
  // 線種、マーカーの指定
  g_def_line(0, G_BLACK, 0, G_LINE_SOLID);
  g_sel_line(0);
  g_def_marker(0, G_RED, 2, G_MARKER_CIRC);
  g_sel_marker(0);

  // 初期値と初期値を置く円を描く
  g_circle(real(g), imag(g), r0, G_YES, G_NO);
  for (i = 0; i < n; i++)
    g_marker(real(x[i]), imag(x[i]));

  // 反復
  for (k = 1; k <= 1000; k++) {
    double error;
    cout << "第" << k << "反復" << endl;
    for (i = 0; i < n; i++) {
      dx[i] = polynomial(n, a, x[i]) / bunbo(n, x, i);
      newx[i] = x[i] - dx[i];
      // 図示する
      g_move(real(x[i]), imag(x[i]));
      g_plot(real(newx[i]), imag(newx[i]));
      g_marker(real(newx[i]), imag(newx[i]));
    }
    // 更新
    for (i = 0; i < n; i++) {
      x[i] = newx[i];
      cout << x[i] << endl;
    }
    // 変化量を計算する (ここは非常に素朴)
    error = 0.0;
    for (i = 0; i < n; i++)
      error += abs(dx[i]);
    cout << "変化量=" << error << endl;
    // 変化量が小さければ収束したと判断して反復を終了する。
    if (error < 1e-12)
      break;
  }

  // GLSC 終了の処理
  cout << "終了には" << endl;
  cout << " グラフィックスのウィンドウをクリック (X の場合)。" << endl;
  cout << " グラフィックスのウィンドウをクローズ (Windows の場合)。" << endl;
  g_sleep(-1.0);
  g_term();

  return 0;
}

// 多項式の値の計算 (Horner 法)
complex<double> polynomial(int n,
      complex<double> *a,
      complex<double> z)
{
  int i;
  complex<double> w;
  w = a[0];
  for (i = 1; i <= n; i++)
    w = (w * z + a[i]);
```

```
  return w;
}

// Π (z i-z j) の計算
// j ≠ i
complex<double> bunbo(int n,
        complex<double> *z,
        int i)
{
  int j;
  complex<double> w(1,0);
  for (j = 0; j < n; j++)
    if (j != i)
      w *= (z[i] - z[j]);
  return w;
}
```

参考文献

[1] Michel, N. and Stoitsov, M.: Fast computation of the Gauss hypergeometric function with all its parameters complex with application to the Pöschl–Teller–Ginocchio potential wave functions, *Computer Physics Communications*, Vol. 178, No. 7, pp. 535–551 (2008).