$9\ldots$ の旅

9.1　複素数の収束

だいぶ以前に少しだけ $0.999\cdots$ の景色を眺めたね。数の旅は整数を中心に見てきたので、すっかり忘れてしまったかな。でも、$0.999\cdots = 1$ だったはずだ。だからどんな整数でも、たとえば $5 = 4.999\cdots$、$-8 = -7.999\cdots$ のように実際は無数の 9 を含んでいるのである。この場合 5 は

$$4.9,\quad 4.99,\quad 4.999,\quad 4.9999,\quad 4.99999,\quad \ldots,\quad 4.999\cdots,\quad \ldots, \to 5$$

と、次第に 9 の数（かず）を増やしながら 5 に近づく数の**極限**になっている。こういう言い方をすると、$4.999\cdots$ は限りなく 5 に近づくのであって、5 とは異なるものという印象を与えてしまうが、間違いなく $4.999\cdots = 5$ である。つまり $4.999\cdots$ は 5 に**収束**するのだ。おっと、ここでは無限の話に深入りするつもりはないよ。それは別の旅で見学してもらいたい。

収束する数は、ちょっとした計算で作れる場合がある。たとえば 0 から始めて、常に 3 を足して 2 で割ることを繰り返してみよう。すると

$$0 \to 1.5 \to 2.25 \to 2.625 \to 2.8125 \to 2.90625 \to \ldots \to 3$$

となることは容易に確かめられる。複素数にもそんなものがないだろうか。

その前に、複素数が発散するってどういうことだろう。実は「限りなく大きい」というのは複素数の発散にはそぐわない。なぜなら、複素数に大小関係は定義できないからだ。理由は、i は $i > 0$ でも $i < 0$ でもないからだ。もちろん 0 でもない。もし、$i > 0$ なら両辺に i を掛けて $i \times i > 0 \times i$ だが、結果は $-1 > 0$ となり矛盾を生ずる。もし、$i < 0$ なら両辺に i を掛けたとき、不等式に負の数を掛けると不等号の向きが変わるので、やはり $-1 > 0$ の矛盾を生む。つまり、複素数に大小なんてないのだ。

では、発散を定義できないかというと、そうでもない。複素数 $z = (a,\ b)$ の大きさを $\sqrt{a^2 + b^2}$ で定義すると、この値は実数だから大小を決められる。要するに、$\sqrt{a^2 + b^2}$ の値が大き方が大きいと決めればよい。

複素数の大きさは**絶対値**と呼び $|z|$ と表す。すぐ気づいたと思うが、$|z|$ は原点からの距離であり、また極形式で表した r の値でもある。このことから複素数の発散とは、原点から限りなく遠くへ離れた状態をさす。

複素数が収束・発散する様子を **C++**で調べてみよう。

programming list [infcomp.cpp]

```
 1: #include <iostream>
 2: #include <complex>
 3:
 4: void recur(std::complex<double> c) {
 5:     std::complex<double> z(0, 0);
 6:
 7:     for(int i = 0; i < 20; i++) {
 8:         z = z * z + c;
 9:         std::cout << "r = " << abs(z) << "; z = " << z << std::endl;
10:     }
11: }
12:
13: int main() {
14:     std::complex<double> c;
15:
16:     std::cout << "input '(a, b)' as a+bi: ";
17:     std::cin >> c;
18:
19:     recur(c);
20:
21:     return 0;
22: }
```

ここでは手軽に `complex` クラスを使わせてもらおう。したがって、入力は `a+bi` ではなく `(a, b)` だ。

まず 13:行目以降を見よう。複素数の入力をして `recur()` 関数を 1 回実行するだけのプログラムだ。ここでは、先の目的地を見据えて関数にしてある。

ところで 5:行目と 14:行目では、複素数の宣言の仕方が少し違うね。なぜだろう。

この違いは、たとえば整数変数 `n` を用いるとき、`int n = 0;` とするか `int n;` とするかの違いだ。変数は宣言と同時に初期化するのが望ましい。しかし複素数型の変数は単に `complex z = (0, 0);` とは書けない。`z` に `(0, 0)` を代入するというより `z` 自体が `(0, 0)` という感覚なのだろう。だから `z(0, 0)` と書く。このとき値 0 の型を示す必要があるので`<double>`が付いているいるのだ。

ということで、14:行目は初期化を省いてみた。直後に `c` へ値が代入されるので問題ないのだが、初期化する習慣は大切だ。

さて `recur()` 関数だが、実数型の複素数を受け取り値は返さない。なぜなら、単に画面に出力するだけの目的だからである。だから `void` 型だ。7:行目から出力が 20 回の繰り返しであることと、出力するのは `z = z * z + c` の結果であることが分かる。`z = z * z + c` は何の計算をしているのだろう。

この計算式を数学っぽく書けば

$$z_{n+1} = z_n^2 + c, \quad z_0 = 0 + 0i, \quad c = a + bi \text{（定数）}$$

ということだ。z_n の書き方は数列を表す**漸化式（ぜんかしき）**で、n に順次 $0, 1, 2, 3, \ldots$ が代入される。初項は $z_0 = 0 + 0i$ である。かりに $c = 1 + i$ とすると、漸化式から $n = 0$ のときの z_1 を

$$z_1 = z_0^2 + c = (0 + 0i)^2 + (1 + i) = 1 + i$$

のように計算できる。次は z_2 で

$$z_2 = z_1^2 + c = (1 + i)^2 + (1 + i) = 1 + 3i$$

となる。そして z_3 は ... と延々続けることになる。

TRY! 続きは、実際にプログラムを実行しよう。入力要求には `(1, 1)` を代入しておこう。

実行結果は、複素数 z_n の大きさ r と、実際の $z_n = (a_n, b_n)$ が表示される。表示される `inf` は `double` 型の上限を超えていることを意味するので、この状態は発散していると見てよい。また、`nan` は `double` 型の下限—0 に近い微小な値—を下回ったことを意味するので、実質的に 0 と思ってよい。

で、いまの計算で分かったことは $c = 1 + i$ にすると、漸化式 $z_{n+1} = z_n^2 + c$, $z_0 = 0$ は発散するということだ。

TRY! いろいろな複素数定数で試してみよ。発散しない複素数が大体どの程度の数が想像できるだろうか。

ところで、プログラム中に漸化式を `z = z * z + c;` と書いたのだが、これでちゃんと複素数の計算をしていることに気づいたかな？　自分で実部・虚部を分けながらコードを書く必要なんかないのである。複素数も実数と同じ式で計算できるなんて、**C++**は気が利いているよね。