3.2 マチンの計略

円周率の値を計算するのに無限級数を用いて **Haskell** に計算させたものの、そこで用いた級数は収束が遅いため計算機向きでないことが分かった。計算機向きの級数に仕立て上げたのはマチン[1]である。もっとも、マチンの時代には計算機などなかった。それでも収束が速い式の方が少ない計算量で精確な値を得られるので、大変重宝する式には違いない。マチンは単純な無限級数に工夫を加えて

$$\frac{\pi}{4} = 4\left(\frac{1}{1\cdot 5} - \frac{1}{3\cdot 5^3} + \frac{1}{5\cdot 5^5} - \cdots\right) - \left(\frac{1}{1\cdot 239} - \frac{1}{3\cdot 239^3} + \frac{1}{5\cdot 239^5} - \cdots\right) \qquad (※)$$

という式をひねり出した。ここでもまた $\dfrac{\pi}{4} = \cdots$ の形である。気になる人にだけ、そっと耳打ちしておこう。グレゴリーの式もマチンの式も $\tan\dfrac{\pi}{4} = 1$ であること、すなわち $\dfrac{\pi}{4} = \arctan 1$ であることが利用されているからだ。そして

$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots \qquad (☆)$$

である。

マチンはこれだけの情報と、ほとんどの人が高校で習うであろう 2 倍角の公式を組み合わせた。まず、$\tan\alpha = \dfrac{1}{5}$ となる角 α があるとする（実際にそのような角は存在し、およそ 11.3° である）。ここに 2 倍角の公式を 2 回適用すると、計算は省略するが $\tan 4\alpha = \dfrac{120}{119}$ が得られる。$\tan\dfrac{\pi}{4} = 1$ なのだから、1 に非常に近い $\dfrac{120}{119}$ が得られる 4α は $\dfrac{\pi}{4}$ に非常に近い角だ。実際、$\tan\left(4\alpha - \dfrac{\pi}{4}\right)$ を加法定理により計算すると $\tan\left(4\alpha - \dfrac{\pi}{4}\right) = \dfrac{1}{239}$ という、比較的小さな値が出る。

このことから

$$\arctan\frac{1}{239} = 4\alpha - \frac{\pi}{4} = 4\arctan\frac{1}{5} - \frac{\pi}{4}$$

であるから、$\dfrac{\pi}{4} = 4\arctan\dfrac{1}{5} - \arctan\dfrac{1}{239}$ が分かった。これに（☆）を適用すると（※）になるのである。

駆け足の説明では理解し難いだろうから、詳しく知りたい人はきちんとした数学の書物を読んでみよう。

さて、いまはマチンの式を

$$\pi = \frac{1}{1}\left(\frac{16}{5} - \frac{4}{239}\right) - \frac{1}{3}\left(\frac{16}{5^3} - \frac{4}{239^3}\right) + \frac{1}{5}\left(\frac{16}{5^5} - \frac{4}{239^5}\right) - \cdots$$

[1] ジョン・マチン（1685–1751）：イギリスの天文学者。

と見て、π の値を計算することにしよう。その上で ( ) のまとまりをひとつの項と考えれば、第 n 項を n の式で表すことができる。それは

$$\text{第 } n \text{ 項} := \pm\frac{1}{2n-1}\left(\frac{16}{5^{2n-1}} - \frac{4}{239^{2n-1}}\right) \quad \text{（奇数番目は+, 偶数番目は } -\text{）}$$

というものであるから、$f(n)$ を再帰的に定義できる。それは

$$f(n) = f(n-1) \pm \frac{1}{2n-1}\left(\frac{16}{5^{2n-1}} - \frac{4}{239^{2n-1}}\right) \quad \text{（奇数番目は+, 偶数番目は } -\text{）}$$

である。ここまで分かれば、あとは `pivalue` 関数と同じように **Haskell** のスクリプトを書けばよい。具体的には、第 n 項の式を差し替えるだけである。

[hs script]

```
pivalue' :: Integer -> Integer
pivalue' 0 = 0
pivalue' n
    | odd  n = pivalue' (n-1) + nth
    | even n = pivalue' (n-1) - nth
    where nth = (16*10^1000 `div` 5^(2*n-1) - 4*10^1000 `div` 239^(2*n-1))
                `div` (2*n-1)
```

はっきり言って不細工なスクリプトである。それでも単に第 n 項の式を差し替えるだけでは、長い式が 2 か所に出て見苦しいので `nth` という項を定義しておいた。`where` 節に使った式は第 n 項そのものだが、長い式なので 2 行に分割して書いてある。入力の際は 1 行で書いてもらいたい。

`10^1000` を使っていることから、小数点以下 1000 桁を想定している。問題は e の計算同様、何項まで計算させればよいかということだ。$\frac{1}{239^{2n-1}}$ はあっというまに 0 に近づくので、$\frac{1}{5^{2n-1}}$ が $\frac{1}{10^{1000}}$ を下回るところを調べればよい。細かいことを言うなら、$\frac{1}{10^{1000}}$ を下回るということは、小数点以下 1000 桁を下回る精度を意味するので、本当は 10^{999} を用いて計算すればよい。でも、そのまま解いてしまおう。それには両辺の**常用対数**をとって

$$\log_{10} 5^{2n-1} > \log_{10} 10^{1000} \quad \text{すなわち} \quad (2n-1)\log_{10} 5 > 1000$$

を解けばよいし、その程度の計算は **Haskell** に任せた方が早い。あらかじめ n を解いておくと $n > \frac{1}{2}\left(\frac{1000}{\log_{10} 5} + 1\right)$ であるから、**Haskell** では

(ghci env.)

```
Prelude> (1000/(logBase 10 5)+1)/2
715.8382790366966
```

とすればよい。単に `log 5` としてしまうと $\log_e 5$ の計算になってしまうので、**底**を a とする対数には `logBase a n` を使う。

さて、とりあえず pivalue' 715 を実行すればよいことが分かった。早速 **Haskell** に計算してもらおう。

(ghci env.)

```
*Main> pivalue' 715
31415926535897932384626433832795028841971693993751058209749445923078164062
86208998628034825342117067982148086513282306647093844609550582231725359408
12848111745028410270193852110555964462294895493038196442881097566593344612
84756482337867831652712019091456485669234603486104543266482133936072602491
41273724587006606315588174881520920962829254091715364367892590360011330530
54882046652138414695194151160943305727036575959195309218611738193261179310
51185480744623799627495673518857527248912279381830119491298336733624406566
43086021394946395224737190702179860943702770539217176293176752384674818467
66940513200056812714526356082778577134275778960917363717872146844090122495
34301465495853710507922796892589235420199561121290219608640344181598136297
74771309960518707211349999998372978049951059731732816096318595024459455346
90830264252230825334468503526193118817101000313783875288658753320838142061
71776691473035982534904287554687311595628638823537875937519577818577805321
71226806613001927876611195909216420199 1
```

ところで、これだと pivalue' に与える引数をあらかじめ求めておく必要があるので、手軽さの点で劣っている。n 桁分の計算がしたければ、引数には計算項数ではなく、桁数 n を与えるのが自然な行為というものだ。しかし、その場合は関数の初期値も 10^n にしておかなくてはならない。pivalue' 関数を活かす形で n 桁の円周率を計算するスクリプトにするなら、こんな感じだろうか。

[hs script]

```
pval' :: Integer -> Integer -> Integer
pval' 0 m = 0
pval' n m
    | odd  n = pval' (n-1) m + nth
    | even n = pval' (n-1) m - nth
    where nth = (16*10^m `div` 5^(2*n-1) - 4*10^m `div` 239^(2*n-1)) `div`
                (2*n-1)

pval :: Integer -> Integer
pval m = pval' (floor((fromIntegral m/(logBase 10 5)+1)/2)) m
```

うわ、これはひどい。pval' 関数は、1000 桁計算限定の pivalue' 関数を m 桁計算のために 10^m に変えてある。ただ、これでは pval' 関数は桁数 m と計算項数 n を受け取らなければならないことになる。よって pval' 関数は、2 個の整数を引数にとり、1 個の値を返す関数となっている。しかし計算項数 n は受け取る必要なんてない。なぜなら、桁数 m を受け取れば計算できるんだから。

そこで、桁数を受け取ったら計算項数を求めて pval' 関数を実行する pval 関数を定義した。これは単純に m から n を計算するための式をそのまま引数に与えるだけの関数である。n の計算は floor((m/(logBase 10 5)+1)/2) だったはずだが、fromIntegralって何だろう。

fromIntegral 関数が必要な理由は、floor 関数が実数値を要求するからである。m だけでは数値の型推論ができないためだ。そこで一旦、m を実数値として扱っている。したがって pval 関数は、整数値を受け取って整数値を返す関数であるが、中では 整数 → 実数 → 整数 と変換しているのである。

これなら手間要らずで円周率の計算ができる。

(ghci env.)

```
*Main> pval 50
31415926535897932384626433832795028841971693993750 9
```

手始めに 50 桁分の計算をしてみた。最後はちょっと誤差があるけど。これなら手軽に 10 万桁でも計算させようという気になるかもしれない。でも、あまり大きな桁数を与えるのはやめよう。マチンのお陰で少し効率よく計算できるようになったとはいえ、相当な計算量になるのだから。