## 3.3 小数点以下の値は？

**Python3** にとって、何百桁もの計算をすることは苦にならない。しかし、$\pi$ の値を何百桁も計算するとなれば話は別だ。大きな数を扱う場合、**配列**を利用して計算する言語は数多くある。**Python3** にはその姉妹版とでも言おうか、数値計算に適した **NumPy** が利用できる。しかし、散歩がてらそれほど大げさなことはしないでおこう。実は、ちょっとした工夫で、円周率の小数点以下の数を何百桁でも求められるのだから。

それは簡単な話だ。たとえば小数点以下 1000 桁の精度で円周率を計算したければ、計算のための数値を $10^{1000}$ の大きさにしてしまえばよい。もちろんそれでは、$3.14\cdots$ ではなく $314\cdots$（1000 桁の数）という値になるので、小数点の位置を改めなくてはならないが。何にしても変更は簡単なので、早速試してみよう。ただし、アルゴリズムに問題があるので 100 桁の計算でやることにする。

[py script]

```
>>> def fxl(n):
...   u = 1; v = 1
...   for n in range(1, n+1):
...     u *= 5
...     v *= 239
...   return (4 * 10**100 // (n * u) - 10**100 // (n * v))
...
>>> def mpil(n):
...   p = 0
...   sgn = 1
...   for n in range(1, n, 2):
...     p += sgn * fxl(n)
...     sgn = -sgn
...   print(4 * p)
...
>>> mpil(300)
31415926535897932384626433832795028841971693993751058209749445923078164062
862089986280348253421170684
```

変更箇所は基本的に `fx` 関数の `return` 文のみである。ここを $10^{100}$ の大きさに変えている。蛇足ながら、関数の中身を変更したので関数名も `fxl` としてある。もうひとつの関数名も `mpil` だ（いずれも long を意味する `l` をつけた）。ここで、小数点をつけて実数計算させる意味はないので、割り算は "`//`" で行った。むしろ大きな意味を持つのは計算回数を決める `n` の値である。まずは、無造作に `300` までの計算とした。しかし、本当はどこまで計算する必要があるんだろう。

対数を使って大雑把に計算しておこう。$\dfrac{1}{(2n-1)5^{2n-1}}$ のほうが $\dfrac{1}{(2n-1)239^{2n-1}}$ より収束が遅いので、100 桁の計算なら $\dfrac{1}{(2n-1)5^{2n-1}}$ が $\dfrac{1}{10^{100}}$ より小さくなれば、その先の計算は不要にな

る。よって

$$\frac{1}{(2n-1)5^{2n-1}} < \frac{1}{10^{100}}$$

を解けばよい。ただし $n$ が十分大きい場合、$(2n-1)$ の桁数は $5^{2n-1}$ の桁数よりはるかに小さい。そこで計算を簡単にするためにも

$$\frac{1}{5^{2n-1}} < \frac{1}{10^{100}}$$

を解けば十分である。

両辺について対数をとると

$$\begin{aligned} -(2n-1)\log 5 &< -100\log 10 \\ 2n-1 &> \frac{100\log 10}{\log 5} \end{aligned}$$

である。$\frac{100\log 10}{\log 5}$ は **Python3** にやってもらおう。

[py script]

```
>>> import math
>>> 100 * math.log(10) / math.log(5)
143.06765580733932
```

数学関数を使うときは、`import math` が必要である。**Python3** を起動してから一度も import していない場合は、忘れずに実行しておこう。結果を見れば、143 項程度の計算で十分ということである。実際、余裕でおつりがくる。

[py script]

```
>>> mpil(143)
31415926535897932384626433832795028841971693993751058209749445923078164062
862089986280348253421170684
```

ところで、不等式を $n$ でなく $2n-1$ で解いたことに注意してもらいたい。スクリプトに使っている `n` は 1, 3, 5, ... と増えている。つまり、$5^{2n-1}$ や $239^{2n-1}$ を `5**n` や `239**n` に見立てているのだ。そのため、計算回数に関わる `n` を求めることは、$2n-1$ の値を求めることになるのである。ただし、100 桁までの値がすべて正確に出ているわけではない。当然、末尾の数桁—この場合は 2 桁—には誤差が生じる。本当に正確な 100 桁の値が知りたければ、`fxl` 関数の数値を `10**105` 程度に増やしておかなくてはならない。

アルゴリズムの問題点についても触れておきたい。それは `fxl` 関数の計算の仕方にある。この関数は $5^k$ や $239^k$ が必要になるたび、正直に 5 や 239 を $k$ 回繰り返し掛けている。加える項が 100 項程度ならまだしも、1000 項、10000 項と増えたとき、このやり方は大きな負担となる。計算式

をよく吟味しよう。もし、$239^{1001}$ の項まで計算が必要だとしたら、それ以前に $239^{999}$ が行われているはずである。これを利用しない手はないだろう。

ただし、ここでは、これ以上スクリプトに手を加えることはしない。再びこの道を通るとき、表示の仕方を工夫するのと一緒に、アルゴリズムの効率も考えることにしよう。