Calculating Pi to the 100th digit

My submission for extra credit question 2

Graham Byrd

May 9, 2022

(This page intentionally left blank)

Contents

1 Overview

In order to calculate π to the 100th digit a bit of research was involved, because we needed a key feature to be present in our code that is not inherently there. We specifically need high precision decimals, as we need to store 100 decimals accurately. We also need an algorithm that is *efficient*, and produces our output in at most a few seconds.

Some searching lead me to the `decimal` library[†] for Python. It had commands that not only allowed me to have high precision decimals, but also allowed me to directly dictate what the precision was, via the `getcontext().prec` variable.

Some more searching lead me to the Chudnovsky Algorithm which currently holds the World Record for most digits of π calculated.

Coding this was somewhat straight forward, and the most work went into figuring out how to take an input of 100 desired digits, and actually translate that to 100 digits calculated.

To quickly show the calculation that we are going to do:

$$\frac{1}{\pi} = 12 \sum_{q=0}^{\infty} \frac{(-1)^q (6q)! (545140134q + 13591409)}{(3q)!(q!)^3 \left(640320\right)^{3q+\frac{3}{2}}} \tag{1.1}$$

Which we will represent more cleanly as:

$$\pi = C \left(\sum_{q=0}^{\infty} \frac{M_q \cdot L_q}{X_q} \right)^{-1} \tag{1.2}$$

Where,

$$C = 426880\sqrt{10005}, \tag{1.3}$$

$$M_q = \frac{(6q)!}{(3q)!(q!)^3}, \tag{1.4}$$

$$L_q = 545140134q + 13591409, \tag{1.5}$$

$$X_q = (-262537412640768000)^q \tag{1.6}$$

This cleaned up function is the principle idea behind my implementation of the algorithm, and the key is to separate the terms that have q in them from the constants.

[†]Documentation can be found here

2 The Code

My implementation of the Chudnovsky algorithm looks like this:

```python
import decimal as dec
import math

def getpi(prec):
        dec.getcontext().prec = prec + 3
        # chudnovsky produces about 14 new digits per iteration,
        # so we define the number of iterations as the desired
        # precision divided by 14 (rounded up)
        q = math.ceil((prec+3) / 14)

        # X, L, and M at iteration 1 are very easy to find, so we
        # skip a loop and make certain parts of the code run faster
        # by starting on iteration 1
        C = 426880 * dec.Decimal(10005).sqrt()
        X = 1
        L = 13591409
        M = 1

        # starting from iteration 1, so our sum is M*L/X = L
        sigma = dec.Decimal(L)

        for i in range(1,q):  # loop for sum
                M = (math.factorial(6*i) //
                        (math.factorial(3*i) * math.factorial(i)**3))
                L += 545140134
                X *= -262537412640768000
                sigma += dec.Decimal(M*L) / X
                pi = C / sigma
        return pi

print(getpi(100))
```

```
[Output:] 3.141592653589793238462643383279502884197169399375105820974944592307816406
  286208998628034825342117067987...
```

This calculated 102 digits, but only 100 of them accurately, and this seems to be an artifact of the algorithm combined with the way the decimal package works. I've given it 100 precision, so it keeps track of decimals to the 103rd currently(note the `prec+3`), but the algorithm doesn't produce 1 digit each loop, in fact, it produces around 14(14.18), so there is a mismatch of precision to digits calculated. That is why I tried to implement the `math.ceil()` command to make up for this, and while it certainly helps, it does not fix the issue. I believe that if our `prec+3` was a multiple of 14, we would see a close or perfect calculation where for a desired precision of n, we get back π accurate to the nth digit.