Computational Methods for π in Python

Thomas Pasfield, Omar Alhomaidah, Owen Mudgett

December 10, 2022

Abstract

Contents

1 Introduction

Pi is an extremely useful number, being fundamental to mathematics for so much more than just geometry, and having extensive applications in physics. It is simply the ratio of the circumference of a circle and the diameter of the circle, yet applies to so much more. Being such a useful number, it is important that we have an accurate value for it, and can calculate it as needed. This paper explores different methods of these calculations, using three categories of methods. These categories are numerical integration, sum of alternating series, and Monte Carlo integration.

2 Problem Statement

The goal is to approximate π with Python using different methods, and to provide an analysis of the methods relative to each other. The code must accept a number of iterations, the "N-value", and run each method using that value.

3 Method/Analysis

3.1 Numerical Integration

For some problems we can't integrate the function. Instead, we need to approximate the integration. So, we use what we call numerical integration. When we are provided a set of data rather than an explicit function or when it is challenging to locate the antiderivative of the integrand, we apply numerical integration. When it is difficult to identify a closed form of an integral or when we simply need an approximation of the integral value, we typically employ numerical integration to estimate its values.

The midpoint rule, trapezoidal rule, and Simpson's rule are the methods for numerical integration that are most frequently used.

Two functions are provided for this task.

$$\int_0^1 4\sqrt{1-x^2}dx$$

$$\int_{-1}^1 \frac{1}{\sqrt{1-x^2}}dx$$

Trapezoid Rule

The Trapezoidal Rule divides the whole area into smaller trapezoids rather than utilizing rectangles to calculate the area under the curves. This integration determines the area by approximating the region underlying a function's graph as a trapezoid. The left and right sums are averaged out according to this rule.

ADD FIGURES

As we see from the figure above, we pick points where Δx intersect and use these values to solve for the trapezoid function.

Trapezoid Function:

$$\int_a^b f(x)dx \approx \Delta \left(\frac{1}{2} f(x_0) + \Sigma_{i=1}^{N-1} f(x_i) + \frac{1}{2} f(x_n) \right)$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x$

Ex.

$$T_n = \frac{\Delta x}{2} \left(f(x_0) + 2f(x_1) + 2f(x_{n-1} + f(x_n) \right)$$

$$T_{10} = \frac{\Delta x}{2} \left(f(0) + 2f(0.1) + 2f(0.2) + \cdots + f(1) \right)$$

$$T_{10} = \frac{0.1}{2} \left(4\sqrt{1-0} 4\sqrt{1-0.1^2} + 4\sqrt{1-0.2^2} + \cdots + 4\sqrt{1-1^2} \right)$$

$$T_{10} = 3.1045$$

Simpson's $\frac{1}{3}$ Rule

The errors for the midpoint and trapezoid rules behave in a highly predictable manner if the function is not linear on a subinterval; they have the opposite sign. For instance, the trapezoid will be too high and the midpoint will be too low if the function is concave up. Thus, it makes reasonable that averaging the trapezoid and midpoint would give a more accurate estimate. However, we can perform better in this instance than a simple average. Using a weighted average will reduce the error. We employ Simpson's, a quadratic polynomial function, to get the appropriate weight.

ADD FIGURES

As we see from the figure above, we will have the same point as trapezoid where we pick points that Δx intersect and use these values to evaluate the Simpson's function.

Simpson's Function:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3}\left(f(x_0) + 4\Sigma_{i=1,3,5}^{N-1} f(x_i) + 2\Sigma_{i=2,4,6}^{N-2} f(x_i) + f(x_n)\right)$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x$

Midpoint Rule

We run the risk of the values at the endpoints not accurately representing the average value of the function on the subinterval if we use the endpoints of the subintervals to approximate the integral. The midpoint of each subinterval is a location that is significantly more likely to be close to the average. The midpoint of every subinterval is used here to calculate the sum.

ADD FIGURES

In the midpoint method, we use the points that occur between Δx values and use these in the Midpoint function to find a solution.

Midpoint Function:

$$\int_a^b f(x)dx \approx \Delta x \Sigma_{i=1}^{N} f(x_i)$$

where $\Delta x = \frac{b-a}{n}$ and $x_i = a + i\Delta x$

Comparison

ADD FIGURES

While the trapezoidal rule uses trapezoidal approximations to estimate the definite integral, the midpoint rule uses rectangular regions to do so. Simpson's rule approximates the original function using quadratic polynomials, then it approximates the definite integral. When the underlying function is smooth, the trapezoidal rule does not provide an accurate number like Simpson's or midpoint rules do. This is due to the fact that Simpson's rule employs quadratic approximation rather than linear approximation. All three methods produce a good approximation for the integrals. Midpoint was the best approximation for our problem, then Simpson's was very close to it and lastly trapezoid was the least accurate method.

3.2 Sum of Alternating Series

Alternating Series

$$\tan^{-1} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \cdots \approx \Sigma_{n=1}^{N}(-1)^{n+1}\frac{x^{2n-1}}{2n-1}$$

The first alternating sum method uses the alternating series of arctan. Multiplying the series by 4 with an x value of 1 over N iterations approximates π. This method is fairly accurate, but has a strange interaction with N values that are exponentiations of 10. Whichever place N occupies - for example, 1000 in the thousands - the digit in the thousandths place in π will be undershot. N = 1000 gives a π value of 3.140592653840, which is accurate up to 3.141592653 except for the thousandths place.

Machin's Formula

$$4\left(4\tan^{-1}(\frac{1}{5}) - \tan^{-1}(\frac{1}{239})\right) = \pi$$

The second alternating sum method uses arctan again, but twice each with different x values. John Machin created the formula in 1706, and has been widely used since then due to how fast the series converges to π (Nishiyama). The first arctan function is multiplied by 16 and uses x = $\frac{1}{5}$, and the second arctan (subtracted from the first) is multiplied by 4 and uses x = $\frac{1}{239}$. Overall, this method is more accurate than normal 4 * arctan(1,N), but takes slightly longer.

Madhava's Series

$$\sqrt{12}\left(1 - \frac{1}{3\cdot 3} + \frac{1}{5\cdot 3^2} - \frac{1}{7\cdot 3^3} + \cdots\right) = \pi$$

he third alternating sum method uses what's called Madhava's Series, with is the square root of 12 multiplied by the the series:

$$\Sigma(-1)^{n+1}/(2n-1)(3^{n-1})$$

starting at n = 1.

This one is faster than the other two, but takes many more iterations before it reaches accurate digits of π.

3.3 Monte Carlo Integration

A Monte Carlo estimator is a method of approximating an explicit value using randomness. They are useful for getting a rough idea of an outcome, rather than getting an exact or accurate one.

Area Method

The ratio between the area of a circle and the area of the square bounding it equals π.

$$\frac{A_{circle}}{A_{square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

This ratio is for each unit of the total area, and is maintained when approximated using randomly distributed points within it.

Volume Method

Like the area method, a ratio can be found relative to pi using a sphere, a cone, and a cube.

$$\frac{(V_{sphere} - V_{cone})}{V_{cube}} = \frac{\left(\frac{4}{3}\pi r^3 - \frac{1}{3}\pi r^2 h\right)}{8r^3} = \frac{\pi}{8}$$

Also like the area method, the ratio is maintained for each unit of volume, so the ratio can be multiplied by the volume of the cube, 8, to find π.

4 Solutions/Results

Our code outputs the following results:

Iterations	Midpoint	Simpson's	Trapezoid	Arctan	Machin's	Madhava	Area	Volume
1	3.464102	1.333333	2.000000	4.000000	3.183264	3.464102	4.000000	4.000000
2	3.259367	2.976068	2.732051	2.666667	3.140597	3.079201	4.000000	5.333333
4	3.183929	3.083595	2.995709	2.895238	3.141592	3.137853	4.000000	3.200000
8	3.156687	3.121189	3.089819	3.017072	3.141593	3.141569	3.111111	1.777778
16	3.146952	3.134398	3.123253	3.079153	3.141593	3.141593	2.588235	1.882353
32	3.143491	3.139052	3.135102	3.110350	3.141593	3.141593	2.909091	2.909091
64	3.142265	3.140695	3.139297	3.125969	3.141593	3.141593	3.200000	2.830769
128	3.141830	3.141275	3.140781	3.133780	3.141593	3.141593	3.162791	3.286822
256	3.141677	3.141481	3.141306	3.137686	3.141593	3.141593	3.143969	3.175097
512	3.141622	3.141553	3.141491	3.139640	3.141593	3.141593	3.150097	3.134503
1000	3.141604	3.141578	3.141555	3.140593	3.141593	3.141593	3.152000	3.248000

5 Discussion/Conclusions

- Sum of Alternating Series using Machin's Formula is the best method for calculating pi, by the metric of how many iterations are required for an extremely accurate result
- All alternating series methods grow exponentially slower as iteration count rises
- Integration method time seems to scale linearly with iteration count, for all methods used
- Monte Carlo scales linearly, very exactly.

Numerical Integration

Numerical Integration is accurate with Equation i., though only achieves 4 correct digits at 1000 iterations. Equation ii is not as accurate, largely due to Simpson's Rule and Trapezoid Rule methods both requiring edge cases. The asymptotes affect calculation.

Sum of Alternating Series

Methods using the sum of an alternating series achieved the value of pi to 12+ decimal places in the least time. Arctan is fast but is 1/1000 off. Madhava's method is slower to calculate and takes more iterations than arctan. Machin's is only slightly slower than arctan but takes significantly less iterations to achieve high accuracy.

Monte Carlo Integration

Monte Carlo methods have the fastest per-iteration time of all the methods, but none of the accuracy of the other methods. Even running at 1,000,000 iterations cannot consistently calculate 5 digits of pi accurately. Volume method takes more calculation time and is more susceptible to large jumps, while area method is faster to calculate and doesn't stray as far from pi at any point.

References

[1] Dawkins, Paul. Calculus II - Approximating Definite Integrals, https://tutorial.math.lamar.edu/classes/calcii/approximatingdefintegrals.aspx.

[2] Libretexts. "1.11: Numerical Integration" Mathematics LibreTexts, Libretexts, 22 Jan. 2022, https://math.libretexts.org

[3] Nishiyama, Yutaka. "Machin's Formula and PI - Personalpages.to.infn.it." Machin's Formula and Pi, 2019, http://personalpages.to.infn.it/ zaninett/pdf/machin.pdf.

A Python Codes

Text introducing this appendix. Subsections and further divisions can also be used in appendices.

```python

#!/usr/bin/env python3
"""
==============================================================================
Title: Examples of Computational Methods for the Approximation of Pi
Team: Team A
Written By: Thomas Pasfield, Omar Alhomaidah, Owen Mudgett
Last Update Date: 12 / 9 / 2022
------------------------------------------------------------------------------
Description:
    This script provides and compares different methods for calculating the
    value of pi, and provides a formatted output in to the terminal.

    Plots of these methods are generated with the accompanying plots.py file.
"""

# Import Sys to allow console inputs
from sys import argv

# Remove this line before submission. Make sure to uncomment your modules
import mcpi as mc
import altsum
import numintegrate as ni

# --------------------------------------------------------------------------
# User input
#   try: and except: are used to ensure a valid input data type. If the input
#   is invalid, it repeats the prompt until the user inputs a valid value.

N = 0  # Added so N always has a defined value
def userInput():
    global N  # N exists outside of function, needs to be global to execute.
    while True:
```

```python
        try:
            N = int(input("N-value? "))
            # Value must also be positive, so throw the same error if it's not.
            if (N <= 0):
                raise ValueError
            break
        except ValueError:
            print("Please input a positive integer. ")
            continue

if len(argv) < 2:
    userInput()
else:
    while True:
        try:
            N = int(argv[1])
            break
        except ValueError:
            print("Run parameter invalid, please correct.")
            userInput()
            break



# ------------------------------------------------------------------------------

# Part 1. Numerical Integration
#   a. The Trapezoid Rule
#   b. The Midpoint Rule

print("--------------------------------------------------------------------------------")
print("Numerical Integration Method with Equation `i.`")
print("  LaTeX:  \int_0^1 4\sqrt{1 - x^2}dx")
print("--------------------------------------------------------------------------------")
print("        N\tMidpoint      \tSimpson's     \tTrapezoid")
i = 0
while 2**i < N:
    n = 2**i
    mid = ni.midpoint_int(ni.f, 0.0, 1.0, n)
    simp = ni.simpson_int(ni.f, 0.0, 1.0, n)
    trap = ni.trapezoid_int(ni.f, 0.0, 1.0, n)

    print(f"{2**i:8}\t{mid:1.12f}\t{simp:1.12f}\t{trap:1.12f}")
```

```
    i += 1

final = [ni.midpoint_int(ni.f, 0, 1, N), ni.simpson_int(ni.f, 0, 1, N), ni.trapezoid_int(ni.f, 0, 1, N)]
print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final[2]:1.12f}")
# print("----------------------------------------")
# print(f"pi = {final:1.16f}, calculated with {N} iterations.")

print()
print("------------------------------------------------------------------------")
print("Numerical Integration Method with Equation `ii.`")
print("  LaTeX:  \int_{-1}^1 \\frac{1}{\sqrt{1 - x^2}}dx")
print("------------------------------------------------------------------------")
print("        N\tMidpoint      \tSimpson's     \tTrapezoid")
i = 0
while 2**i < N:
    n = 2**i
    mid = ni.midpoint_int(ni.g, -1.0, 1.0, n)
    simp = ni.simpson_int(ni.g, -1.0, 1.0, n)
    trap = ni.trapezoid_int(ni.g, -1.0, 1.0, n)

    print(f"{2**i:8}\t{mid:1.12f}\t{simp:1.12f}\t{trap:1.12f}")
    i += 1

final = [ni.midpoint_int(ni.g, -1, 1, N), ni.simpson_int(ni.g, -1, 1, N), ni.trapezoid_int(ni.g, -1, 1, N)]
print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final[2]:1.12f}")
# print("----------------------------------------")
# print(f"pi = {final:1.16f}, calculated with {N} iterations.")

print()



# ---------------------------------------------------------------------------

# PART 2. Sum of Alternating Series

print()
print("------------------------------------------------------------------------")
```

```python
print("Alternating Series Methods")
print("------------------------------------------------------------------------")
print("        N\tarctan        \tMachin        \tMadhava")
i = 0
while 2**i < N:
    n = 2**i
    arc = 4 * altsum.arctan(1, n)
    machin = altsum.machine(n)
    madhava = altsum.madhava(n)

    print(f"{2**i:8}\t{arc:1.12f}\t{machin:1.12f}\t{madhava:1.12f}")
    i += 1

final = [4*altsum.arctan(1, N), altsum.machine(N), altsum.madhava(N)]
print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final[2]:1.12f}")
# print("----------------------------------------")
# print(f"pi = {final:1.16f}, calculated with {N} iterations.")

print()



# ------------------------------------------

# Part 3. Monte Carlo Integration
#   a. Area Method

area_pi_iterated = mc.mc_area_v(N)
area_pi = area_pi_iterated[-1]
vol_pi_iterated = mc.mc_volume_v(N)
vol_pi = vol_pi_iterated[-1]

print("------------------------------------------------------------------------")
print("Monte Carlo Method: Pi using area of a circle")
print("------------------------------------------------------------------------")
print("        N\tarea            \tvolume")
i = 0
while 2**i < N:
```

```
    print(f"{2**i:8}\t{area_pi_iterated[2**i]:1.12f}\t{vol_pi_iterated[2**i]:1.12f}")
    i += 1

print(f"{N:8}\t{area_pi:1.12f}\t{vol_pi:1.12f}")
# print("----------------------------------------")
# print(f"pi = {area_pi:1.5f}, calculated with {N} iterations.")

print()
"""
#   b. Volume Method
vol_pi_iterated = mc.mc_volume_v(N)
vol_pi = vol_pi_iterated[-1]
print("--------------------------------------------------------------------------")
print("Monte Carlo Method: Pi using Volume of Sphere & Cone")
print("--------------------------------------------------------------------------")
print("        N\tpi")
i = 0
while 2**i < N:
    print(f"{2**i:8}\t{vol_pi_iterated[2**i]:1.5f}")
    i += 1

print(f"{N:8}\t{vol_pi:1.5f}")
print("----------------------------------------")
print(f"pi = {vol_pi:1.5f}, calculated with {N} iterations.")

print()
"""
```

```

#!/usr/bin/env python3
"""
==========================================================================================
Title: Numerical Intergration Calculation Methods for Pi
Team: Team A
Written By: Omar Alhomaidah
Last Update Date: 12 / 8 / 2022
------------------------------------------------------------------------------------------
Description:
"""

# Imports
from math import sqrt
```

```

# Define the functions to represent the equations of the definite integrals
# i
def f(x):
    y = 4.0*sqrt(1.0 - x**2)
    return y
# ii
def g(x):
    # Catches the asymptotes, doesn't let them affect calculation
    # Edited by Thomas so it works for Simpson's and Trapezoid Rules
    # (Blame Thomas if this is incorrect, this is written by Thomas)
    if x**2 == 1.0:
        return 0
    y = 1.0/sqrt(1.0 - x**2)
    return y


# Integration Methods:
# Midpoint rule:
def midpoint_int(eq, a, b, N):
    dx = (b-a)/N
    m_sum = 0
    for i in range(N):
        m_sum += eq((a + dx/2) + i*dx)
    m_sum *= dx
    return m_sum

# Simpson's 1/3rd rule:
def simpson_int(eq,a,b,N):
    dx = (b-a)/N
    s_sum = eq(a) + eq(b)
    for i in range(1,N):
        xi_bar = a+(i*dx)

        if i%2 == 0:
            s_sum += 2*eq(xi_bar)
        else:
            s_sum += 4*eq(xi_bar)
    return (dx/3)*s_sum


# Trapezoid rule:
def trapezoid_int(eq,a,b,N):
    dx = (b-a)/N
    t_sum = eq(a) + eq(b)
```

```
    for i in range(1,N):
        xi_bar = a+(i*dx)

        t_sum += 2*eq(xi_bar)
    return (dx/2)*t_sum
```

```

"""
==============================================================================
 Title: Sum of Alternating Series Methods to Calculate Pi
Team: Team A
Written By: Owen Mudgett
Last Update Date: 12 / 9 / 2022
------------------------------------------------------------------------------
 Description:
    Approximates pi using an alternating series definition of arctan. Three
    methods are used, including just arctan, Machin's Formula, and Madhava's
    Series.

    arctan(x, N):
        This method takes x as the input, and N as the iteration limit for
        calculating the arctan of that input.

    machine(N):
        Input N is iteration limit to be fed into arctans.
        Computes pi using two arctans in a formula.
        See https://en.wikipedia.org/wiki/Machin-like_formula for general info
            about how this method works

    madhava(N):
        Series method for calculating pi without an trig operations.
        Accepts N as input for iteration limit.
        https://en.wikipedia.org/wiki/Madhava_series
            Add `#Another_formula_for_the_circumference_of_a_circle` to
            navigatev directly to the proper section of the article.
"""

# Imports
from math import sqrt

```

```
# a. arctan function part
def arctan(x,N):
    xsum=0
    for i in range(N):
        i+=1
        if i > N:
            break
        xsum += ((-1)**(i+1)) * (x**(2*i-1)) / (2*i-1)
    # Removed the 4 multiplier because it needs to be general purpose
    return xsum

# b. Machin's Formula
def machine(N):
    machins_pi = 16*arctan(1/5, N) - 4*arctan(1/239, N)
    return machins_pi

# c. Madhava Series
def madhava(N):
    xsum=0
    for i in range(N):
        i+=1
        if i > N:
            break
        xsum += ((-1)**(i+1))/((2*i-1)*(3**(i-1)))
    return sqrt(12) * xsum


# -- General Info --
# Alternating Series
# 4 * arctan(N)
# This works, but undershoots the value by magnitude of 10^x
# For example, N=1,000,000 will give accurate values up to the hundred
#   thousandths digit, but will undershoot the millionths digit slightly

# Machin's Formula:
# Is very accurate, getting the first 6 digits at N = 10 and above

# Madhava's Series:
# Is very accurate, gets 6 beginning digits at N = 10 and more than Spyder can display correct at N = 100
```

```

"""
==========================================================================
```

```
Title: Monte Carlo Method for the Approximation of Pi
Team: Team A
Written By: Thomas Pasfield
Last Update Date: 12 / 7 / 2022
--------------------------------------------------------------------------------
Description:
    Provides two methods to approximate pi. mc_area() uses the area of a circle
    inside a square region, mc_volume() uses the overlap of a sphere and cone in
    a rectangular prism shaped region.

    Each function mentioned above return a double value, which is the last
    approximation calculated.

    mc_area_v() returns an array of values reached during execution.
    Same with mc_volume_v().
"""

# Imports
import random

# AREA METHODS

# Generates random values within the volume for x, and y.
# Input: None
# Output: 2 doubles
def area_points():
    x = random.uniform(-1,1)
    y = random.uniform(-1,1)
    return x,y

# Approximates pi using a monte carlo method by checking if points are within
#   a circle within a square area
# Input: N (Iteration Limit)
# Output: pi (final approximation)
def mc_area(N):
    # Random point generation
    count = 0

    for i in range(N):
        x,y = area_points()
        if x*x + y*y < 1.0:
            count += 1

```

```
    r = count / N
    return 4 * r

# Approximates pi using a monte carlo method by checking if
    points are within
#   a circle within a square area. Returns the approximated
    value for every
#   iteration.
# Input: N (Iteration Limit)
# Output: [pi_0, pi_1, ... pi_N]
def mc_area_v(N):
    # Random point generation
    vals = []
    count = 0

    for i in range(N):
        x,y = area_points()
        if x*x + y*y < 1.0:
            count += 1
        r = count / (i+1)
        vals.append(r*4)
    return vals



# VOLUME METHODS

# Generates random values within the volume for x, y, and z.
# Input: None
# Output: 3 doubles
def vol_points():
    x = random.uniform(-1,1)
    y = random.uniform(-1,1)
    z = random.uniform( 0,2)
    return x,y,z

# Approximates pi using a monte carlo method by checking if
    points are within
#   a sphere and cone intersection.
# Input: N (Iteration Limit)
# Output: pi (final approximation)
def mc_volume(N):
    count = 0
    for i in range(N):
        x,y,z = vol_points()
        if x*x + y*y < z*z and x*x + y*y + (z-1)**2 < 1:
            count += 1
    r = count / N
    return r*8
```

```
# Approximates pi using a monte carlo method by checking if points are within
#   a sphere and cone intersection. Returns the approximated value for every
#   iteration.
# Input: N (Iteration Limit)
# Output: [pi_0, pi_1, ... pi_N]
def mc_volume_v(N):
    vals = []
    count = 0
    for i in range(N):
        x,y,z = vol_points()
        if x*x + y*y < z*z and x**2 + y**2 + (z-1)**2 < 1:
            count += 1
        r = count / (i+1)
        vals.append(r*8)

    return vals
```