

*MA305, Fall 2022*

*Embry-Riddle Aeronautical University*

---

## Computational Methods for $\pi$ in Python

---

Thomas Pasfield, Omar Alhomaiah, Owen Mudgett

December 10, 2022

## Abstract

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Problem Statement</b>	<b>3</b>
<b>3</b>	<b>Method/Analysis</b>	<b>3</b>
3.1	Numerical Integration . . . . .	3
3.2	Sum of Alternating Series . . . . .	6
3.3	Monte Carlo Integration . . . . .	7
<b>4</b>	<b>Solutions/Results</b>	<b>7</b>
<b>5</b>	<b>Discussion/Conclusions</b>	<b>8</b>
<b>A</b>	<b>Python Codes</b>	<b>10</b>

## 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  $\pi$  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  $\Delta x$  intersect and use these values to solve for the trapezoid function.

Trapezoid Function:

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

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

Ex.

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

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

$$T_{10} = \frac{0.1}{2} (4\sqrt{1-0.04}\sqrt{1-0.1^2} + 4\sqrt{1-0.2^2} + \cdots + 4\sqrt{1-1^2})$$

$$T_{10} = 3.1045$$

## 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  $\Delta x$  intersect and use these values to evaluate the Simpson's function.

Simpson's Function:

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

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  $\Delta x$  values and use these in the Midpoint function to find a solution.

Midpoint Function:

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

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

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

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  $\pi$ . 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  $\pi$  will be undershot. N = 1000 gives a  $\pi$  value of 3.140592653840, which is accurate up to 3.141592653 except for the thousandths place.

### Machin's Formula

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

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  $\pi$  (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

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

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  $\pi$ .

### 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  $\pi$ .

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

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  $\pi$ .

## 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/calci/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.

```
1
2 #!/usr/bin/env python3
3 """
4
5 Title: Examples of Computational Methods for the Approximation
6       of Pi
7 Team: Team A
8 Written By: Thomas Pasfield , Omar Alhomaiah, Owen Mudgett
9 Last Update Date: 12 / 9 / 2022
10
11 Description:
12     This script provides and compares different methods for
13     calculating the
14     value of pi, and provides a formatted output in to the
15     terminal.
16
17     Plots of these methods are generated with the accompanying
18     plots.py file.
19 """
20
21 # Import Sys to allow console inputs
22 from sys import argv
23
24 # Remove this line before submission. Make sure to uncomment
25 # your modules
26 import mcpic as mc
27 import altsum
28 import numintegrate as ni
29
30 # -----
31
32 # User input
33 # try: and except: are used to ensure a valid input data type.
34 # If the input
35 # is invalid, it repeats the prompt until the user inputs a
36 # valid value.
37
38 N = 0 # Added so N always has a defined value
39 def userInput():
40     global N # N exists outside of function, needs to be global
41     to execute.
42     while True:
```

```

34         try:
35             N = int(input("N-value? "))
36             # Value must also be positive, so throw the same
error if it's not.
37             if (N <= 0):
38                 raise ValueError
39             break
40         except ValueError:
41             print("Please input a positive integer. ")
42             continue
43
44 if len(argv) < 2:
45     userInput()
46 else:
47     while True:
48         try:
49             N = int(argv[1])
50             break
51         except ValueError:
52             print("Run parameter invalid, please correct.")
53             userInput()
54             break
55
56
57
58 # -----
59
60 # Part 1. Numerical Integration
61 #   a. The Trapezoid Rule
62 #   b. The Midpoint Rule
63
64 print("
65
66 ")
67 print("Numerical Integration Method with Equation 'i.'")
68 print("  LaTeX:  \int_0^1 4\sqrt{1 - x^2}dx")
69
70
71
72
73
74
75
76 print(f" {2*i:8}\t{mid:1.12f}\t{simp:1.12f}\t{trap:1.12f}")

```

```

77     i += 1
78
79 final = [ni.midpoint_int(ni.f, 0, 1, N), ni.simpson_int(ni.f, 0,
80               1, N), ni.trapezoid_int(ni.f, 0, 1, N)]
81 print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final
82       [2]:1.12f}")
83 # print("_____")
84 # print(f"pi = {final:1.16f}, calculated with {N} iterations.")
85
86 print()
87 print("
88 print("
89
90     ")
91     N\tMidpoint      \tSimpson's      \tTrapezoid")
92     i = 0
93     while 2**i < N:
94         n = 2**i
95         mid = ni.midpoint_int(ni.g, -1.0, 1.0, n)
96         simp = ni.simpson_int(ni.g, -1.0, 1.0, n)
97         trap = ni.trapezoid_int(ni.g, -1.0, 1.0, n)
98
99         print(f"{2**i:8}\t{mid:1.12f}\t{simp:1.12f}\t{trap:1.12f}")
100        i += 1
101
102 final = [ni.midpoint_int(ni.g, -1, 1, N), ni.simpson_int(ni.g,
103               -1, 1, N), ni.trapezoid_int(ni.g, -1, 1, N)]
104 print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final
105       [2]:1.12f}")
106 # print("_____")
107 # print(f"pi = {final:1.16f}, calculated with {N} iterations.")
108
109 print()
110
111 # -----
112 # -----
113
114 # PART 2. Sum of Alternating Series
115
116 print()
117 print("
118
119 print()

```

```

115 print("Alternating Series Methods")
116 print("
    ")
117 print("          N\tarctan          \tMachin          \tMadhava")
118 i = 0
119 while 2**i < N:
120     n = 2**i
121     arc = 4 * altsum.arctan(1, n)
122     machin = altsum.machine(n)
123     madhava = altsum.madhava(n)
124
125     print(f"{2**i:8}\t{arc:1.12f}\t{machin:1.12f}\t{madhava:1.12f}")
126     i += 1
127
128 final = [4*altsum.arctan(1, N), altsum.machine(N), altsum.madhava(N)]
129 print(f"{N:8}\t{final[0]:1.12f}\t{final[1]:1.12f}\t{final[2]:1.12f}")
130 # print("-----")
131 # print(f"pi = {final:1.16f}, calculated with {N} iterations.")
132
133 print()
134
135
136
137 # -----
138
139 # Part 3. Monte Carlo Integration
140 #   a. Area Method
141
142 area_pi_iterated = mc.mc_area.v(N)
143 area_pi = area_pi_iterated[-1]
144 vol_pi_iterated = mc.mc_volume.v(N)
145 vol_pi = vol_pi_iterated[-1]
146
147 print("
    ")
148 print("Monte Carlo Method: Pi using area of a circle")
149 print("
    ")
150 print("          N\tarea          \tvolume")
151 i = 0
152 while 2**i < N:

```

```

153     print(f"{2**i:8}\t{area_pi_iterated[2**i]:1.12f}\t{
vol_pi_iterated[2**i]:1.12f}")
154     i += 1
155
156 print(f"{N:8}\t{area_pi:1.12f}\t{vol_pi:1.12f}")
157 # print("_____")
158 # print(f"pi = {area_pi:1.5f}, calculated with {N} iterations.")
159
160 print()
161 """
162 #    b. Volume Method
163 vol_pi_iterated = mc.mc_volume_v(N)
164 vol_pi = vol_pi_iterated[-1]
165 print
166     ("_____")
167
168 print("Monte Carlo Method: Pi using Volume of Sphere & Cone")
169 print
170     ("_____")
171
172 print("          N\tpi")
173 i = 0
174 while 2**i < N:
175     print(f"{2**i:8}\t{vol_pi_iterated[2**i]:1.5f}")
176     i += 1
177
178 print(f"{N:8}\t{vol_pi:1.5f}")
179 print("_____")
180 print(f"pi = {vol_pi:1.5f}, calculated with {N} iterations.")
181
182 print()
183 """
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

15
16 # Define the functions to represent the equations of the
    definite integrals
17 # i
18 def f(x):
19     y = 4.0*sqrt(1.0 - x**2)
20     return y
21 # ii
22 def g(x):
23     # Catches the asymptotes, doesn't let them affect
    calculation
24     # Edited by Thomas so it works for Simpson's and Trapezoid
    Rules
25     # (Blame Thomas if this is incorrect, this is written by
    Thomas)
26     if x**2 == 1.0:
27         return 0
28     y = 1.0/sqrt(1.0 - x**2)
29     return y
30
31
32 # Integration Methods:
33 # Midpoint rule:
34 def midpoint_int(eq, a, b, N):
35     dx = (b-a)/N
36     m_sum = 0
37     for i in range(N):
38         m_sum += eq((a + dx/2) + i*dx)
39     m_sum *= dx
40     return m_sum
41
42 # Simpson's 1/3rd rule:
43 def simpson_int(eq,a,b,N):
44     dx = (b-a)/N
45     s_sum = eq(a) + eq(b)
46     for i in range(1,N):
47         xi_bar = a+(i*dx)
48
49         if i%2 == 0:
50             s_sum += 2*eq(xi_bar)
51         else:
52             s_sum += 4*eq(xi_bar)
53     return (dx/3)*s_sum
54
55
56 # Trapezoid rule:
57 def trapezoid_int(eq,a,b,N):
58     dx = (b-a)/N
59     t_sum = eq(a) + eq(b)

```

```

60     for i in range(1,N):
61         xi_bar = a+(i*dx)
62
63         t_sum += 2*eq(xi_bar)
64     return (dx/2)*t_sum

```

---

```

1
2 """
3
4 Title: Sum of Alternating Series Methods to Calculate Pi
5 Team: Team A
6 Written By: Owen Mudgett
7 Last Update Date: 12 / 9 / 2022
8
9 Description:
10     Approximates pi using an alternating series definition of
11     arctan. Three
12     methods are used, including just arctan, Machin's Formula,
13     and Madhava's
14     Series.
15
16     arctan(x, N):
17         This method takes x as the input, and N as the iteration
18         limit for
19         calculating the arctan of that input.
20
21     machine(N):
22         Input N is iteration limit to be fed into arctans.
23         Computes pi using two arctans in a formula.
24         See https://en.wikipedia.org/wiki/Machin-like\_formula
25         for general info
26         about how this method works
27
28     madhava(N):
29         Series method for calculating pi without an trig
30         operations.
31         Accepts N as input for iteration limit.
32         https://en.wikipedia.org/wiki/Madhava\_series
33         Add '#
34         Another_formula_for_the_circumference_of_a_circle' to
35         navigatev directly to the proper section of the
36         article.
37 """
38
39 # Imports
40 from math import sqrt

```



```

35 # a. arctan function part
36 def arctan(x,N):
37     xsum=0
38     for i in range(N):
39         i+=1
40         if i > N:
41             break
42         xsum += ((-1)**(i+1)) * (x**(2*i-1)) / (2*i-1)
43     # Removed the 4 multiplier because it needs to be general
44     # purpose
45     return xsum
46
47 # b. Machin's Formula
48 def machine(N):
49     machins_pi = 16*arctan(1/5, N) - 4*arctan(1/239, N)
50     return machins_pi
51
52 # c. Madhava Series
53 def madhava(N):
54     xsum=0
55     for i in range(N):
56         i+=1
57         if i > N:
58             break
59         xsum += ((-1)**(i+1)) / (((2*i-1)*(3*(i-1)))
60     return sqrt(12) * xsum
61
62 # — General Info —
63 # Alternating Series
64 # 4 * arctan(N)
65 # This works, but undershoots the value by magnitude of 10^x
66 # For example, N=1,000,000 will give accurate values up to the
67 # hundred
68 # thousandths digit, but will undershoot the millionths digit
69 # slightly
70
71 # Machin's Formula:
72 # Is very accurate, getting the first 6 digits at N = 10 and
73 # above
74
75 # Madhava's Series:
76 # Is very accurate, gets 6 beginning digits at N = 10 and more
77 # than Spyder can display correct at N = 100

```

```

1
2 """
3

```

```

4 Title: Monte Carlo Method for the Approximation of Pi
5 Team: Team A
6 Written By: Thomas Pasfield
7 Last Update Date: 12 / 7 / 2022
8
9 Description:
10     Provides two methods to approximate pi. mc_area() uses the
11     area of a circle
12     inside a square region, mc_volume() uses the overlap of a
13     sphere and cone in
14     a rectangular prism shaped region.
15
16     Each function mentioned above return a double value, which
17     is the last
18     approximation calculated.
19
20     mc_area_v() returns an array of values reached during
21     execution.
22     Same with mc_volume_v().
23 """
24
25 # Imports
26 import random
27
28 # AREA METHODS
29
30 # Generates random values within the volume for x, and y.
31 # Input: None
32 # Output: 2 doubles
33 def area_points():
34     x = random.uniform(-1,1)
35     y = random.uniform(-1,1)
36     return x,y
37
38 # Approximates pi using a monte carlo method by checking if
39 # points are within
40 # a circle within a square area
41 # Input: N (Iteration Limit)
42 # Output: pi (final approximation)
43 def mc_area(N):
44     # Random point generation
45     count = 0
46
47     for i in range(N):
48         x,y = area_points()
49         if x*x + y*y < 1.0:
50             count += 1

```

```

47     r = count / N
48     return 4 * r
49
50 # Approximates pi using a monte carlo method by checking if
51 #   points are within
52 #   a circle within a square area. Returns the approximated
53 #   value for every
54 #   iteration.
55 # Input: N (Iteration Limit)
56 # Output: [pi_0, pi_1, ... pi_N]
57 def mc_area_v(N):
58     # Random point generation
59     vals = []
60     count = 0
61
62     for i in range(N):
63         x,y = area_points()
64         if x*x + y*y < 1.0:
65             count += 1
66             r = count / (i+1)
67             vals.append(r*4)
68     return vals
69
70 # VOLUME METHODS
71
72 # Generates random values within the volume for x, y, and z.
73 # Input: None
74 # Output: 3 doubles
75 def vol_points():
76     x = random.uniform(-1,1)
77     y = random.uniform(-1,1)
78     z = random.uniform( 0,2)
79     return x,y,z
80
81 # Approximates pi using a monte carlo method by checking if
82 #   points are within
83 #   a sphere and cone intersection.
84 # Input: N (Iteration Limit)
85 # Output: pi (final approximation)
86 def mc_volume(N):
87     count = 0
88     for i in range(N):
89         x,y,z = vol_points()
90         if x*x + y*y < z*z and x*x + y*y + (z-1)**2 < 1:
91             count += 1
92     r = count / N
93     return r*8

```

```

93
94 # Approximates pi using a monte carlo method by checking if
    points are within
95 #   a sphere and cone intersection. Returns the approximated
    value for every
96 #   iteration.
97 # Input: N (Iteration Limit)
98 # Output: [pi_0, pi_1, ... pi_N]
99 def mc_volume_v(N):
100     vals = []
101     count = 0
102     for i in range(N):
103         x,y,z = vol_points()
104         if x*x + y*y < z*z and x**2 + y**2 + (z-1)**2 < 1:
105             count += 1
106         r = count / (i+1)
107         vals.append(r*8)
108
109     return vals

```