

JF COMPUTATIONAL LABORATORY REPORT SHEET

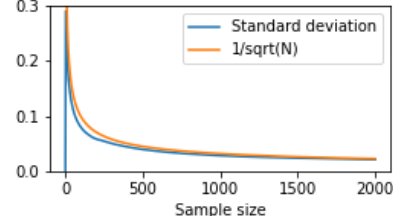
Monte Carlo Methods

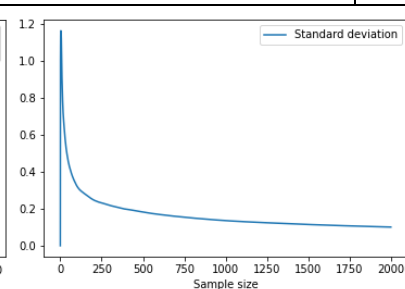
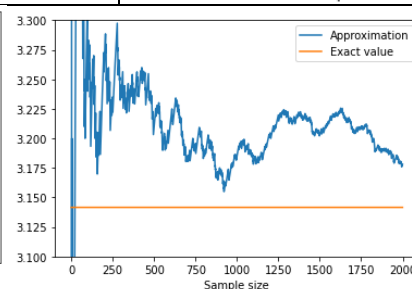
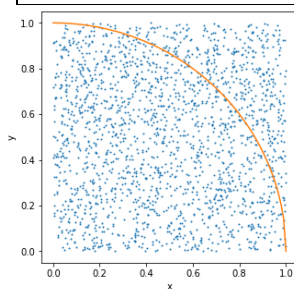
*This should be filled in using Word submitted as a **hardcopy with graphs and commented source code**. Please don't forget to include your name and student number.*

CP2: Monte Carlo	Week: 7
Student Name: Dominik Kuczyński	Date: 10.11.2021
Laboratory Number : 1	Student number: 21367544

Abstract (no more than 200 words) In this report, the results and observations regarding using the Monte Carlo method are given. The method was used to calculate numerical problems: calculate π from an area of a circle, as well as calculate the definite integral of various functions.	[4]
---	-----

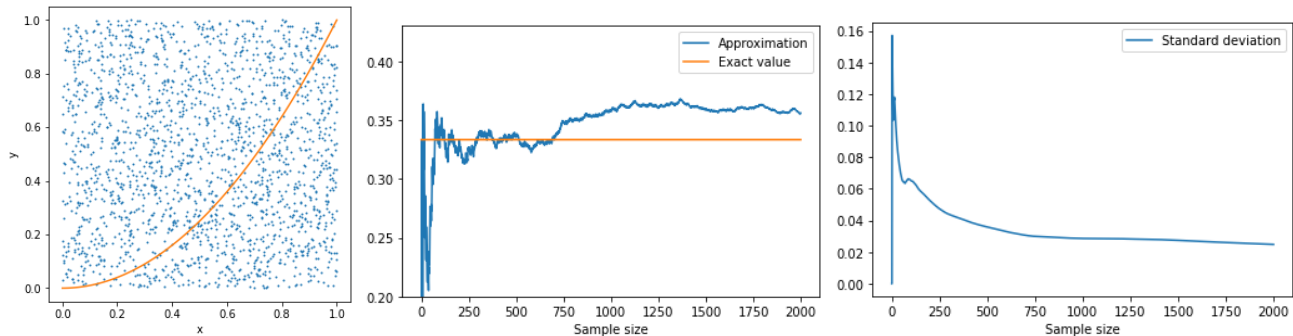
Section 1

What makes the Monte Carlo method different from other methods for solving numerical problems?	The Monte Carlo method does not depend at all on the complexity of the given problem, and the procedure is independent of the given function. Also, the result is always only an approximation.	[1]
Explain why our expression for π will be the same if we only use the upper right quadrant:	Ratio of circle segment area to square area: $R = \frac{1}{4} \pi r^2 / r^2 = \frac{\pi}{4}$	[1]
How does your approximate value π_{approx} compare to the exact value for π ?	$\pi_{\text{approx}} = 3.178$ error = $ \pi - \pi_{\text{approx}} \approx 0.0364$	[1]
How does the error decrease with the number of samples?	The error decreases very quickly at first, but with more samples, it does so increasingly slowly	[1]
Does the error fit some function of N ? If so, what is your guess for the function?	$f(N) \approx \frac{1}{\sqrt{N}}$, as the standard error of the mean in a binomial distribution. 	[1]



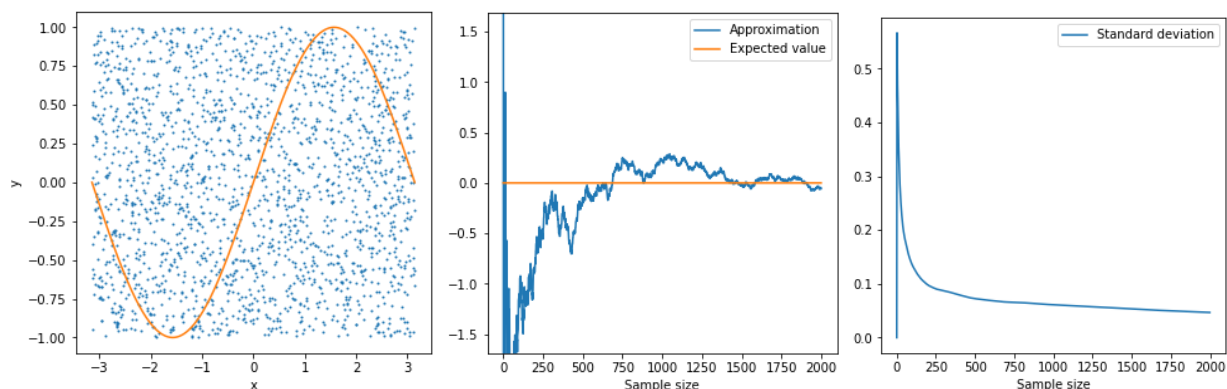
Section 2

What is the exact value of the definite integral I ?	Integral, $I = \int_0^1 x^2 dx = x^3 _0^1 = \frac{1}{3}$	
How does your approximate value I_{approx} compare to the exact value for I ?	$I_{\text{approx}} = 0.343$ error = $ I - I_{\text{approx}} \approx 0.01$	[1]
Does the error decrease in the same way as for our approximation of π ?	Yes, the error seems to be decreasing in the same way (in accordance with the binomial distribution)	[1]



Section 3

What is your function?	$f(x) = \sin(x)$ on $(-\pi, \pi)$	
What is the exact value of your definite integral I ?	Integral, $I = \int_{-\pi}^{\pi} \sin(x) dx = \cos(x) _{-\pi}^{\pi} = -1 - (-1) = 0$	
How does your approximate value I_{approx} compare to the exact value for I ?	$I_{\text{approx}} = -0.101$ error = $ I - I_{\text{approx}} = 0.101$	[1]
Does the error decrease in the same way as it did in part 2? If so, what does this tell you about the Monte Carlo Method?	The error decreases in the same way as in part 1 and 2. This implies that no matter the complexity of the function, the Monte Carlo method will perform about the same with the same number of points.	[1]
Could this numerical integration method be easily extended to multiple dimensions? If so, how?	This method of numerical integration could easily be extended to multiple dimensions by generating points in \mathbb{R}^n as tuples of n random numbers and integrating multi-variable functions.	[1]



Task 1 : circle

```
1. # File: circle.py
2. # Python laboratory - week 7
3. # Monte Carlo method (task 1)
4. # author - Dominik Kuczynski
5.
6. import matplotlib.pyplot as plt
7. import numpy as np
8.
9. circle_x = np.linspace(0, 1, 100)
10. circle_y = np.sqrt(1-circle_x**2)
11. # lists of x and y coordinates:
12. # circle_x -> 100 evenly spaced numbers in (0,1)
13. # circle_y -> respective y-coordinates of points on the circle with center
14. #           at (0,0) and radius 1.
15.
16.
17. fig1 = plt.figure()
18. fig1.set_size_inches(5, 5)
19. # a square figure is created
20.
21. times = 2000
22. # a variable for determining the amount of points scattered on the graph and
23. # used for approximation
24.
25. x = np.random.rand(times)
26. y = np.random.rand(times)
27. # x,y - lists of random coordinates of the scattered points
28.
29. plt.plot(x, y, 'o', markersize=1)
30. plt.plot(circle_x, circle_y)
31. plt.xlabel("x")
32. plt.ylabel("y")
33. # the points are plotted on a new graph
34.
35. dist_squared = x**2 + y**2
36. incircle = dist_squared <= 1
37. # a boolean determining whether each point is inside the circle.
38. # This is equivalent to np.sqrt(x**2+y**2)<=1 but faster computationally.
39.
40. incircle_ratio = float(np.sum(incircle))/float(times)
41. # the ratio of points in the circle to all points
42.
43. pi = incircle_ratio * 4
44. # as we're only looking at a quarter of the circle, the achieved value has
45. # to be multiplied by 4.
46. print('The estimated value of pi is: ', pi)
47.
48. cumulative_incircle = np.cumsum(incircle)
49. # these are the numbers of points inside the circle, depending on the number
50. # of points scattered.
51. # cumulative_incircle[x] = how many out of x points landed in the circle
52.
53. cumulative_ratios = (cumulative_incircle /
54.                      np.arange(1, times+1, dtype=np.float))
55. # the ratios of points inside the circle to all points, depending on number
56. # of points.
57.
58. pis = cumulative_ratios * 4
59. # the respective predicted values of pi
60.
61. plt.figure()
62. approx_pis, = plt.plot(pis)
63. pi, = plt.plot(np.repeat(np.pi, times))
64.
65. plt.ylim(3.1, 3.3)
66. plt.xlabel("Sample size")
```

```

67. plt.title("Approximation vs Sample Size")
68. plt.legend([approx_pis, pi], ["Approximation", "Exact value"])
69. # the predicted values of pi are plotted against the total number of points
70. # scattered. The plot is scaled from 3.1 to 3.3 on the y-axis to better show
71. # the improvement of predictions.
72.
73. def cummean(arr):
74.     return np.cumsum(arr) / np.arange(1, len(arr)+1, dtype=np.float)
75. # a function for calculating the cumulative means of elements in an array up to
76. # every index
77.
78. def cumstd(arr):
79.     return np.sqrt(cummean(arr**2) - cummean(arr)**2)
80. # a function for calculating the cumulative standard deviation of elements in
81. # an array up to every index
82.
83. plt.figure()
84. stdevs, = plt.plot(cumstd(pis))
85. plt.legend([stdevs], ["Standard deviation"])
86. plt.xlabel("Sample size")
87. plt.title("Standard Deviation vs Sample Size")
88. # the cumulative standard deviation is plotted against the number of samples

```

Task 2: parabola

```

1. # file: parabola.py
2. # Python laboratory - week 7
3. # Monte Carlo method (task 2)
4. # author - Dominik Kuczynski
5.
6. import matplotlib.pyplot as plt
7. import numpy as np
8.
9. parabola_x = np.linspace(0, 1, 100)
10. parabola_y = parabola_x**2
11. # lists of x and y coordinates:
12. # parabola_x -> 100 evenly spaced numbers in (0,1)
13. # parabola_y -> respective y-coordinates of points on graph of f(x)=x**2
14.
15.
16. fig1 = plt.figure()
17. fig1.set_size_inches(5, 5)
18. # a square figure is created
19.
20. times = 2000
21. # a variable for determining the amount of points scattered on the graph and
22. # used for approximation
23.
24. x = np.random.rand(times)
25. y = np.random.rand(times)
26. # x,y - lists of random coordinates of the scattered points
27.
28. plt.plot(x, y, 'o', markersize=1)
29. plt.plot(parabola_x, parabola_y)
30. plt.xlabel("x")
31. plt.ylabel("y")
32. # the points are plotted on a new graph
33.
34. test_function = y <= x**2
35. # a boolean determining whether each point is below the graph.
36.
37. below_ratio = float(np.sum(test_function))/float(times)
38. # the ratio of points below the graph to all points
39.
40. print('The estimated value of the integral is: ', below_ratio)
41.
42. cumulative_below = np.cumsum(test_function)
43. # these are the numbers of points below the graph, depending on the number

```

```

44. # of points scattered.
45. # cumulative_below[x] = how many out of x points landed below the graph
46.
47. cumulative_ratios = (cumulative_below /
48.                      np.arange(1, times+1, dtype=np.float))
49. # the ratios of points below the graph to all points, depending on number
50. # of points.
51. # As the area of the square in which all points are scattered is equal to 1,
52. # these are also the approximated integrals.
53.
54. plt.figure()
55. approx_integrals, = plt.plot(cumulative_ratios)
56. integral, = plt.plot(np.repeat(1.0/3.0, times))
57.
58. plt.ylim(0.2, 0.43)
59. plt.xlabel("Sample size")
60. plt.title("Approximation vs Sample Size")
61. plt.legend([approx_integrals, integral], ["Approximation", "Exact value"])
62. # the predicted values of the integral are plotted against the total number of
63. # points scattered. The plot is scaled from 0.2 to 0.43 on the y-axis to better
64. # show the improvement of predictions.
65.
66. def cummean(arr):
67.     return np.cumsum(arr) / np.arange(1, len(arr)+1, dtype=np.float)
68. # a function for calculating the cumulative means of elements in an array up to
69. # every index
70. def cumstd(arr):
71.     return np.sqrt(cummean(arr**2) - cummean(arr)**2)
72. # a function for calculating the cumulative standard deviation of elements in
73. # an array up to every index
74.
75. plt.figure()
76. stdevs, = plt.plot(cumstd(cumulative_ratios))
77. plt.legend([stdevs], ["Standard deviation"])
78. plt.xlabel("Sample size")
79. plt.title("Standard Deviation vs Sample Size")
80. # the cumulative standard deviation is plotted against the number of samples

```

Task 3: sin(x)

```

1. # file: anyf.py
2. # Python laboratory - week 7
3. # Monte Carlo method (task 3)
4. # author - Dominik Kuczynski
5.
6. # For this task, I have taken a flexible approach - this means that
7. # (as far as I know), basically any function as well as any interval may be
8. # used to approximate with the monte carlo method.
9.
10. import matplotlib.pyplot as plt
11. import numpy as np
12.
13. #####
14. def f(x):
15.     return np.sin(x)
16.
17. xlower_bound = -np.pi
18. xupper_bound = np.pi
19.
20. times = 200000
21. expected_value = 0.0
22.
23. # Above are values to be input by the user: the function f of which an interval
24. # is to be approximated, the bounds of the interval as well as the number of
25. # samples and expected value.
26. #####
27.
28. def sign(x):

```

```

29.     return x/np.abs(x)
30. # this is a function returning -1 if x<0 and 1 if x>0
31.
32. def cummean(arr):
33.     return np.cumsum(arr) / np.arange(1, len(arr)+1, dtype=np.float)
34. def cumstd(arr):
35.     return np.sqrt(cummean(arr**2) - cummean(arr)**2)
36. # these are two functions described in the previous task, returning the
37. # cumulative mean and standard deviation of an array.
38.
39.
40. x_plot = np.linspace(xlower_bound, xupper_bound,
41.                      int((xupper_bound-xlower_bound)*100))
42. y_plot = f(x_plot)
43. # x_plot - 100 points spaced evenly between the bounds of the integral
44. # y_plot - the corresponding values of the function f
45.
46.
47. fig1 = plt.figure()
48. fig1.set_size_inches(5, 5)
49. # a square figure is created
50.
51.
52. ylower_bound = np.min(y_plot)
53. yupper_bound = np.max(y_plot)
54. if(ylower_bound > 0 and yupper_bound > 0):
55.     ylower_bound = 0
56. if(ylower_bound < 0 and yupper_bound < 0):
57.     yupper_bound = 0
58. # bounds for the y coordinates are created from the maximum and minimum
59. # values of the function on the interval, making sure to include
60. # the x-axis i.e. y=0.
61.
62. x_random = np.random.rand(times)*(xupper_bound-xlower_bound) + xlower_bound
63. y_random = np.random.rand(times)*(yupper_bound-ylower_bound) + ylower_bound
64. # "times" random points are generated with random coordinates starting at
65. # the lower bounds and up to the upper bounds.
66.
67. plt.plot(x_random, y_random, 'o', markersize=1)
68. plt.plot(x_plot, y_plot)
69. plt.xlabel("x")
70. plt.ylabel("y")
71. # the random points as well as the function are plotted
72.
73.
74. box_size = (xupper_bound-xlower_bound)*(yupper_bound-ylower_bound)
75. # to correctly approximate the area below the graph,
76. # we need to know the overall area of the square, i.e. box_size.
77.
78. below = sign(y_random) * (np.abs(y_random) <= np.abs(f(x_random)))
79. # a value indicating whether a point is closer to zero than a value of f
80. # if not closer to zero than f --> 0
81. # if closer to zero than f & above y=0 --> 1
82. # if closer to zero than f & below y=0 --> -1
83. # this allows for calculating the area below y=0 as negative
84.
85. below_ratio = np.sum(below) / times
86. # a ratio of points included in integral to all scattered points
87. approximation = box_size * below_ratio
88. print("The estimated value of the integral is: ", approximation)
89. # from the ratio, the area below the graph is calculated and printed
90.
91.
92. cumulative_ratio = np.cumsum(below) / np.arange(1, times+1,
93.                                                  dtype=np.float)
94. cumulative_integral = box_size * cumulative_ratio
95. # the cumulative ratios of points below the graph to all points are
96. # calculated, and from them, the respective approximated areas below the graph
97.
98. fig2 = plt.figure()

```

```

99. fig2.set_size_inches(5,5)
100. # another square figure is created
101.
102. achieved, = plt.plot(cummulative_integral)
103. expected, = plt.plot(np.repeat(expected_value, times))
104. # the achieved approximations and the expected value are plotted against
105. # sample size
106.
107. plt.ylim(expected_value-cumstd(cummulative_integral)[int(0.05*times)],
108.           expected_value+cumstd(cummulative_integral)[int(0.05*times)])
109. # to make the plot legible for an arbitrary function, its y-axis is scaled
110. # from expected_value-delta to expected_value+delta, with delta being the
111. # standard deviation of the first 5% of approximations.
112.
113. plt.legend([achieved, expected], ["Approximation", "Expected value"])
114. plt.xlabel("Sample size")
115. plt.title("Approximation vs Sample Size")
116.
117. fig3 = plt.figure()
118. fig3.set_size_inches(5, 5)
119. # a third square figure is created
120.
121. stdevs, = plt.plot(cumstd(cummulative_ratio))
122. plt.legend([stdevs], ["Standard deviation"])
123. plt.xlabel("Sample size")
124. plt.title("Standard Deviation vs Sample Size")
125. # the cummulative standard deviation is plotted against sample size
126.

```

Additional marks:

Tidy workbook	[1]
Graph of π for part 1	[2]
Graph of integral for part 2	[2]
Graph of integral for part 3	[1]