**Digital BH-loop measurement setup and algorithm**

A pair of electronic devices with wires

Description automatically generated

Contents

[Measurement setup 2](#_Toc148535298)

[Selection of the scales for H and B 5](#_Toc148535299)

[Appendix: algorithms in Python 8](#_Toc148535300)

[Wheatstone calculator 8](#_Toc148535301)

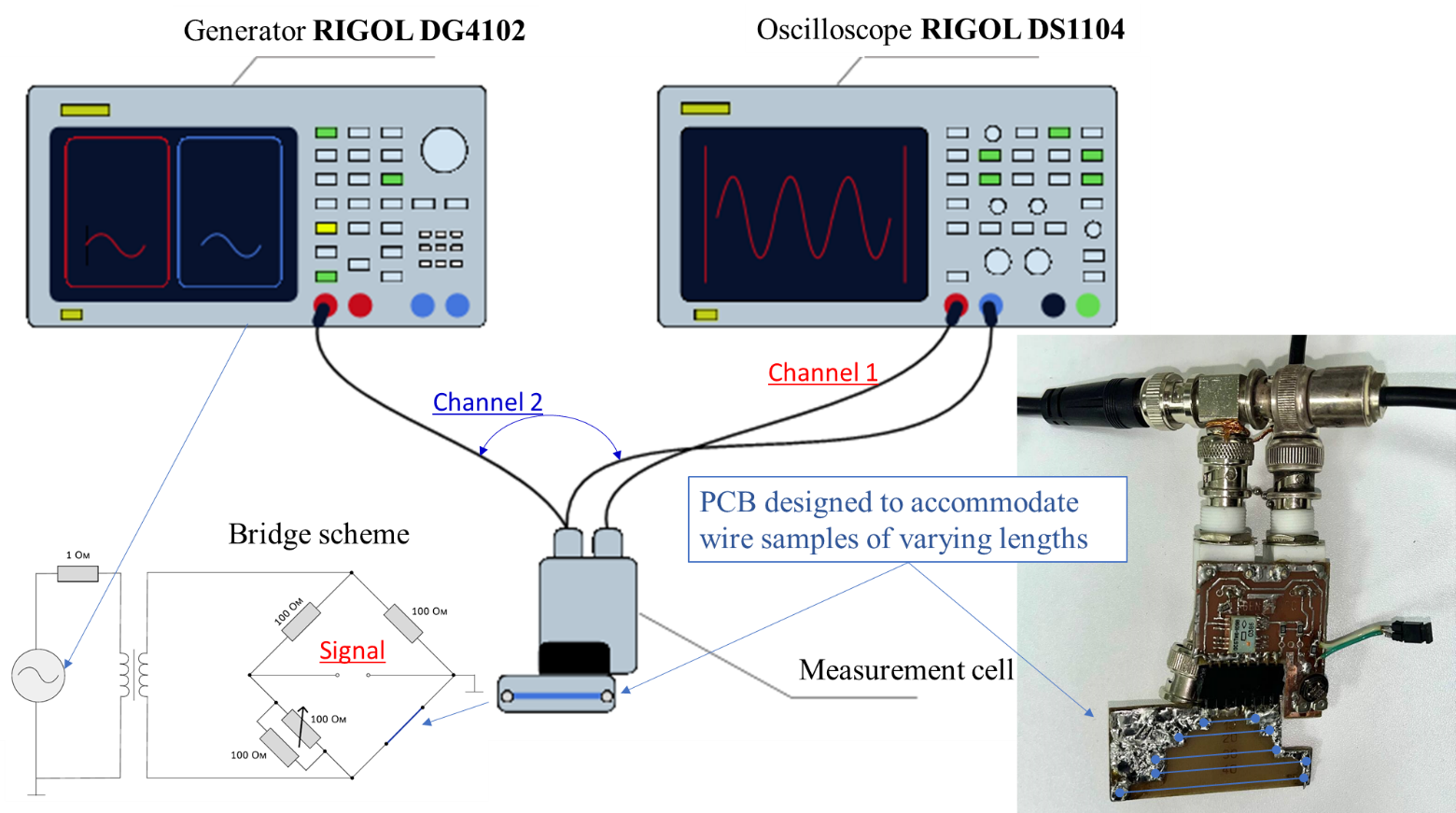
[BH-loop algorithm 11](#_Toc148535302)

**Project package:** [**https://github.com/DYK-Team/Digital\_BH-loop\_algorithm**](https://github.com/DYK-Team/Digital_BH-loop_algorithm)

# Measurement setup

This project investigates the circular magnetisation of amorphous ferromagnetic microwires induced by alternating current. The circular processes and circular magnetic permeability play a pivotal role in influencing the sensitivity of impedance variations concerning external factors, such as magnetic field or mechanical stress. Consequently, this study holds significance in both fundamental understanding and practical applications, particularly in the development of magnetic sensors that operate based on [the magneto-impedance effect](https://pubs.aip.org/aip/apl/article-abstract/65/9/1189/64067/Magneto-impedance-effect-in-amorphous-wires?redirectedFrom=fulltext) (MI).

The diagram of the measurement setup is shown in Fig. 1. A microwire sample constituted one of the arms of [a Wheatstone bridge circuit](https://en.wikipedia.org/wiki/Wheatstone_bridge) powered from a RIGOL DG4102 sinusoidal voltage generator. The recorded signals were measured synchronously on a 2-channel oscilloscope RIGOL DS1104. The first channel measured the voltage of the exciting sinusoid (reference signal). While the signal from the output of the bridge circuit was supplied to the second channel. Since the generator and oscilloscope are connected to opposite diagonals of the bridge, and their screens are connected through a common ground loop, if there is no galvanic isolation on the generator side, a short circuit will occur in the bridge arm. A miniature high-frequency transformer was used as a galvanic isolation. The bridge circuit was balanced using a tuning resistor in such a way as to highlight the potential difference at the ends of the microwire, induced by an alternating circular flux of the magnetic induction inside the wire during magnetization reversal of the circular domains.



**Fig. 1.** Measurement setup.

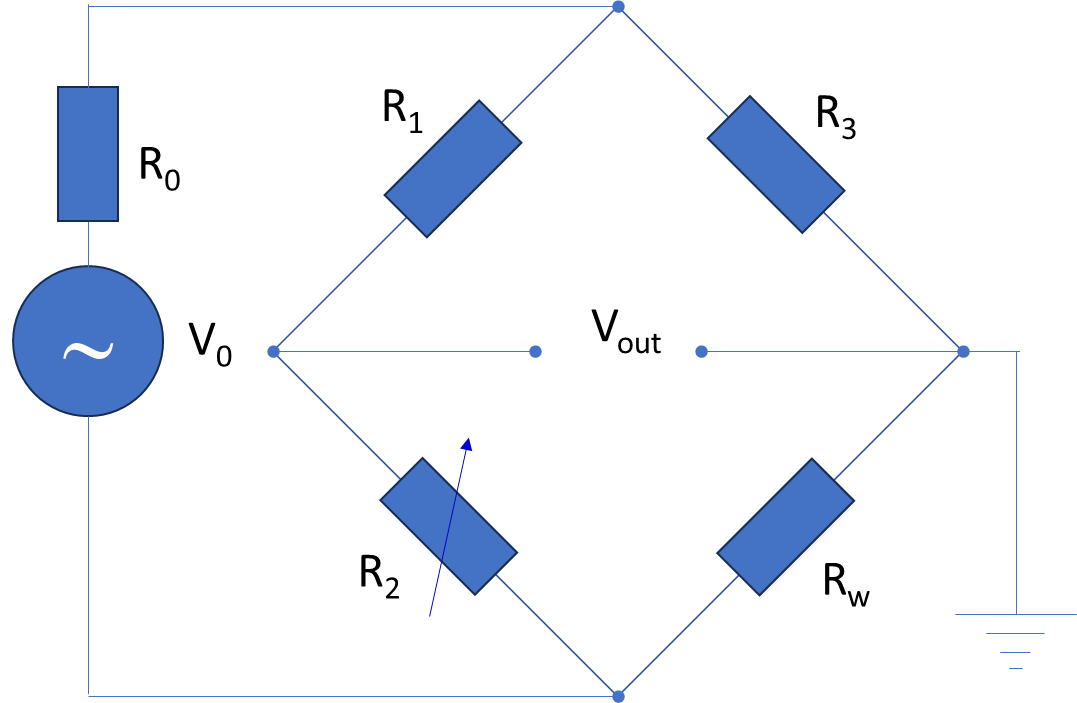
Let us consider the classical [Wheatstone bridge circuit](https://en.wikipedia.org/wiki/Wheatstone_bridge) in Fig. 2, where all the elements are purely resistive without any magnetic properties. Always pay attention to where the ground is established in the circuit. In our case, it is where the outer shell of the coax is connected. The internal resistance of the voltage source equals − in or case this is the resistance of the transformer secondary coil. The balancing, when the output voltage is zero, requires the following condition:

(1)

where is the value of the tuning resistor and is the microwire DC resistance. The general equation for the potential difference across is as follows:

(2)

We require this transfer function to establish a scale for the magnetizing field . We have developed a simple Wheatstone calculator in Python for calculating transfer functions in a balanced bridge circuit (see [Appendix](#_Wheatstone_calculator)). Additionally, you can utilize the circuit in the [LTspice simulator](https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html) (saved in the “Wheatstone\_Calculator\_LTspice\_Vw\_V0” project folder), with its interface shown in Fig. 3.



**Fig. 2.** Wheatstone bridge circuit.

A screenshot of a computer

Description automatically generated

**Fig. 3.** LTspice circuit for calculating the transfer function. .

Balancing the bridge, when the resistor is substituted with a ferromagnetic microwire of the same DC resistance, results in the detection of an additional electromotive force (emf) . This emf is induced by a circular magnetic flux during the magnetization reversal of circular magnetic domains. The balanced bridge circuit, now devoid of an external source but featuring the additional source , is illustrated in Fig. 4.



**Fig. 4.** The balanced bridge circuit where the resistor is substituted with a ferromagnetic microwire of the same DC resistance.

While will be the measured parameter, is essential for calculating the BH-loops. Consequently, we have established the transfer function (see Wheatstone calculator in [Appendix](#_Whitestone_calculator)):

(3)

Additionally, you can utilize the circuit in the [LTspice simulator](https://www.analog.com/en/design-center/design-tools-and-calculators/ltspice-simulator.html) (saved in the “Wheatstone\_Calculator\_LTspice\_VM\_Vout” project folder), with its interface shown in Fig. 5.

A screenshot of a computer

Description automatically generated

**Fig. 5**. LTspice circuit for calculating the inverse transfer function. .

# Selection of the scales for H and B

The following rationale aims to propose a reasonable selection of scales for the magnetizing field and magnetic induction . Let us examine [Maxwell's equation](https://en.wikipedia.org/wiki/Maxwell%27s_equations) for Ampere's circuital law in its integral form (SI units):

(4)

where represents the vector of current density for free charges across a surface , is the contour of edge traversed in the positive direction (with the contour being on the left), denotes the scalar product of the magnetizing field and the unit vector , which is tangent in the positive direction to at each integration point. Meanwhile, is a unit vector normal to the surface at each integration point, and its direction is determined according to the gimlet rule while rotating in the positive direction along the traversal of . Lastly, represents the scalar product of the displacement vector and .

In Eq. (4), we can safely disregard the displacement vector in comparison to the conduction current, which is an accurate assumption for thin microwires and the frequencies at which we measure their hysteresis BH-loops. By selecting a partial microwire cross-section with a variable radius ( is the microwire radius) as the integration surface , we can represent the surface integral over the current density as follows:

(5)

where is the full current through the microwire. Due to the cylindrical symmetry of the microwire, the contour integral on the left side of Eq. (4) can be expressed in the following manner:

(6)

where is the circular field amplitude at the radius . Therefore, the field inside the wire as a function of the radius can be calculated as follows:

(7)

Unlike the measurement scheme for longitudinal hysteresis loops, this field exhibits significant inhomogeneity along the radius. Hence, the average value should be considered as a representative measure of the field intensity. Therefore, we propose the following value for the amplitude of the magnetizing field (SI units):

(8)

The dimensions are indicated within square brackets. Here, should be calculated from , which is measured on the second channel of the oscilloscope (reference sinusoid), following Eq. (2). Note that represents the DC resistance of the microwire.

Now, we shall establish a measurement scale for magnetic induction by utilizing [the Faraday-Maxwell equation](https://en.wikipedia.org/wiki/Maxwell%27s_equations) in its integral form (SI units):

(9)

where is the vector of electrical field, is the vector of magnetization, and is the vacuum permeability. We select the longitudinal section of the microwire passing along its axis as the integration surface, as shown in Fig. 6. Consequently, the integration contour will traverse the microwire surface, with the neglect of end effects. The first non-magnetic term contributing to the emf is compensated by the bridge circuit. By performing contour integration from the left, we obtain:

(10)

where is the voltage induced by the magnetization reversal, and is the voltage measured on the first channel of the oscilloscope (see Eq. (3)). Both areas (see Fig. 6), where and are the wire radius and length respectively. The surface integrals on the right-hand side of Eq. (10), representing the magnetic induction flux , have dimensions in Webers (Wb). To obtain the result in Teslas (T), both sides should be normalized by the integration area :

(11)

Hence, the magnetic induction averaged over half of the longitudinal section () can be reconstructed through time integration:

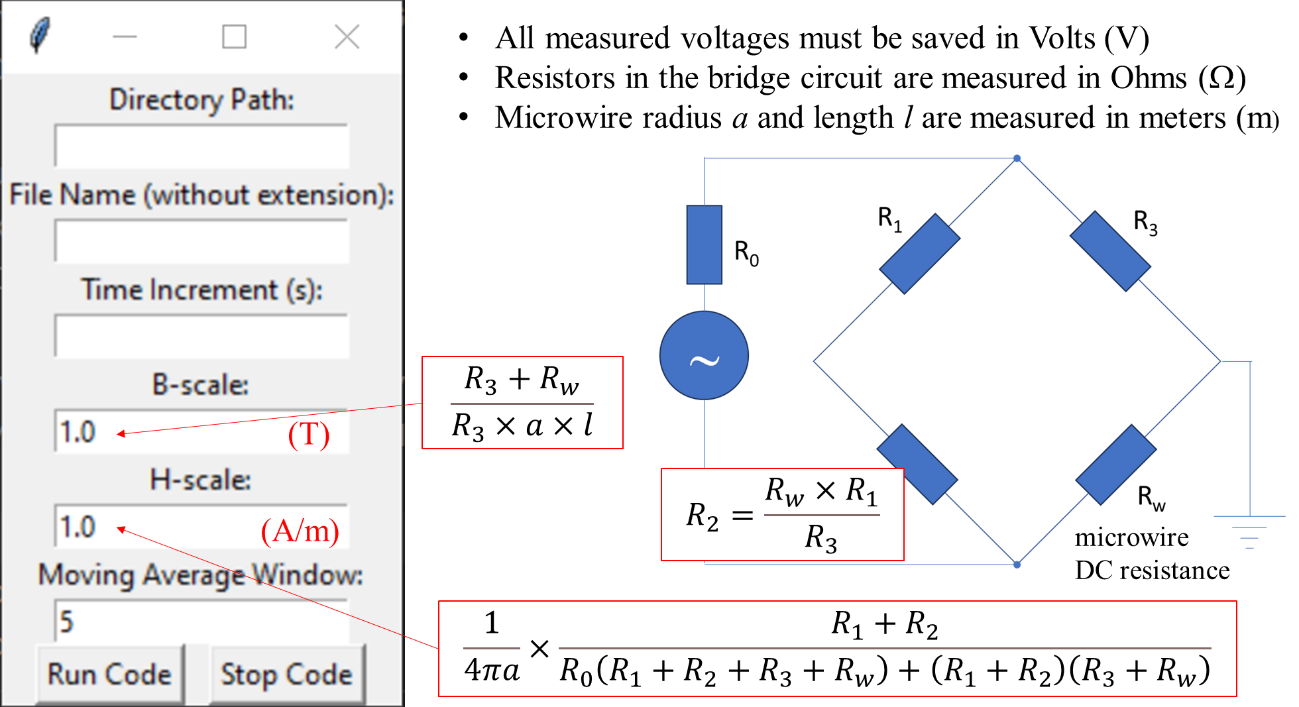
(12)

The dimensions are indicated within square brackets. The voltage output from the bridge circuit is measured on the first channel of the oscilloscope and should be expressed in Volts (V). We have developed an algorithm for integrating and reconstructing the hysteresis BH-loop, provided in the [Appendix](#_BH-loop_algorithm).



**Fig. 6.** Longitudinal section of the microwire representing the integration geometry for Eq. (9).

Using Eqs. (2), (3), (8), and (12), let us establish the rules for calculating the dimensional coefficients for the [program's GUI](#_BH-loop_algorithm), as shown in Fig. 7. By default, both the B and H scales are set to 1.



**Fig. 7.** Rules for calculating the dimensional coefficients.

# Appendix: algorithms in Python

## Wheatstone calculator

This GUI algorithm calculates two transfer functions: and , which are necessary to establish scales for the magnetizing field and magnetic induction .

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Wheatstone.py (with GUI):**

1. #

2. # Wheatstone calculator

3. # Project repository on GitHub: https://github.com/DYK-Team/Digital\_BH-loop\_algorithm

4. # 16/10/2023

5. #

6. # Authors:

7. # Ekaterina Nefedova, Dr. Mark Nemirovich, Dr. Nikolay Udanov, and Prof. Larissa Panina

8. # MISiS, Moscow, Russia, https://en.misis.ru/

9. #

10. # Dr. Dmitriy Makhnovskiy

11. # DYK+ team, United Kingdom, www.dykteam.com

12. #

13.

14. import tkinter as tk

15. from PIL import Image, ImageTk

16.

17. def calculate():

18. R0 = float(R0\_entry.get())

19. R1 = float(R1\_entry.get())

20. R3 = float(R3\_entry.get())

21. Rw = float(Rw\_entry.get())

22.

23. # Calculate R2 that provides the balance of the bridge circuit

24. R2 = R1 \* Rw / R3

25.

26. # Calculate VM / Vout

27. VM\_Vout = (R3 + Rw) / R3

28.

29. # Calculate Vw / V0

30. Vw\_V0 = ((R1 + R2) \* Rw) / (R0 \* (R1 + R2 + R3 + Rw) + (R1 + R2) \* (R3 + Rw))

31.

32. # Update the equation label with the calculated R2 value and the value for VM / Vout and Vw / V0

33. equation\_label.config(text=f"R2 = Rw x R1 / R3 = {R2:.3f} Ohms\n\nVM / Vout = (R3 + Rw) / R3 = {VM\_Vout:.3f}"

34. f"\n\nVw / V0 = {Vw\_V0:.3f}")

35.

36. # Function to stop the calculation and close the program

37. def stop\_calculation():

38. window.quit()

39.

40. # Create the main window

41. window = tk.Tk()

42. window.title("Resistor Calculator")

43.

44. # Create a frame for the left side

45. left\_frame = tk.Frame(window)

46. left\_frame.grid(row=0, column=0, padx=10, pady=10) # Increased space using pady

47.

48. # Create a frame for the right side (picture)

49. right\_frame = tk.Frame(window)

50. right\_frame.grid(row=0, column=1, padx=10, pady=10) # Increased space using pady

51.

52. # Add a message for entering resistor values

53. message\_label = tk.Label(left\_frame, text="Enter the resistor values in Ohms:")

54. message\_label.grid(row=0, columnspan=2, pady=10) # Increased space using pady

55.

56. # Create labels and entry fields for resistor values with increased space

57. R0\_label = tk.Label(left\_frame, text="R0:")

58. R0\_label.grid(row=1, column=0, pady=10) # Increased space using pady

59. R0\_entry = tk.Entry(left\_frame)

60. R0\_entry.grid(row=1, column=1, pady=10) # Increased space using pady

61.

62. R1\_label = tk.Label(left\_frame, text="R1:")

63. R1\_label.grid(row=2, column=0, pady=10) # Increased space using pady

64. R1\_entry = tk.Entry(left\_frame)

65. R1\_entry.grid(row=2, column=1, pady=10) # Increased space using pady

66.

67. R3\_label = tk.Label(left\_frame, text="R3:")

68. R3\_label.grid(row=3, column=0, pady=10) # Increased space using pady

69. R3\_entry = tk.Entry(left\_frame)

70. R3\_entry.grid(row=3, column=1, pady=10) # Increased space using pady

71.

72. Rw\_label = tk.Label(left\_frame, text="Rw:")

73. Rw\_label.grid(row=4, column=0, pady=10) # Increased space using pady

74. Rw\_entry = tk.Entry(left\_frame)

75. Rw\_entry.grid(row=4, column=1, pady=10) # Increased space using pady

76.

77. # Create a label to display the R2 and Vw / Vout equations with increased space

78. equation\_label = tk.Label(left\_frame, text="R2 = Rw x R1 / R3\n\nVM / Vout = (R3 + Rw) / R3\n\nVw / V0 = ...")

79. equation\_label.grid(row=5, columnspan=2, pady=20) # Increased space using pady

80.

81. # Create buttons to trigger the calculation and stop the calculation with increased space

82. calculate\_button = tk.Button(left\_frame, text="Calculate", command=calculate)

83. calculate\_button.grid(row=6, column=0, pady=10) # Increased space using pady

84.

85. stop\_button = tk.Button(left\_frame, text="STOP", command=stop\_calculation)

86. stop\_button.grid(row=6, column=1, pady=10) # Increased space using pady

87.

88. # Create a label to display the result

89. result\_label = tk.Label(left\_frame, text="")

90. result\_label.grid(row=7, columnspan=2, pady=10) # Increased space using pady

91.

92. # Open and resize the PNG image using Pillow without any filters

93. image = Image.open("Circuit.png")

94. new\_width = 600 # Adjust the desired width for a larger picture

95. aspect\_ratio = float(new\_width) / float(image.width)

96. new\_height = int(image.height \* aspect\_ratio)

97. image = image.resize((new\_width, new\_height))

98.

99. photo = ImageTk.PhotoImage(image)

100. image\_label = tk.Label(right\_frame, image=photo)

101. image\_label.photo = photo # To prevent the image from being garbage collected

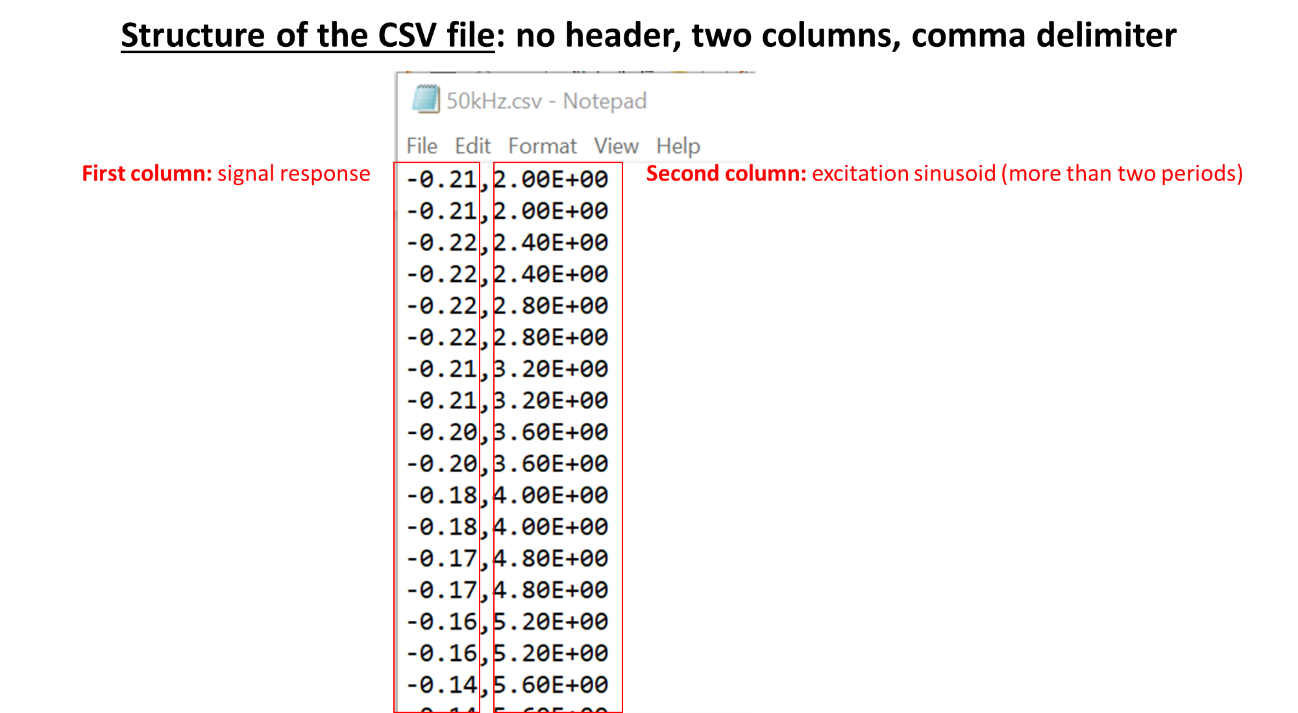
102. image\_label.grid(row=0, column=0)

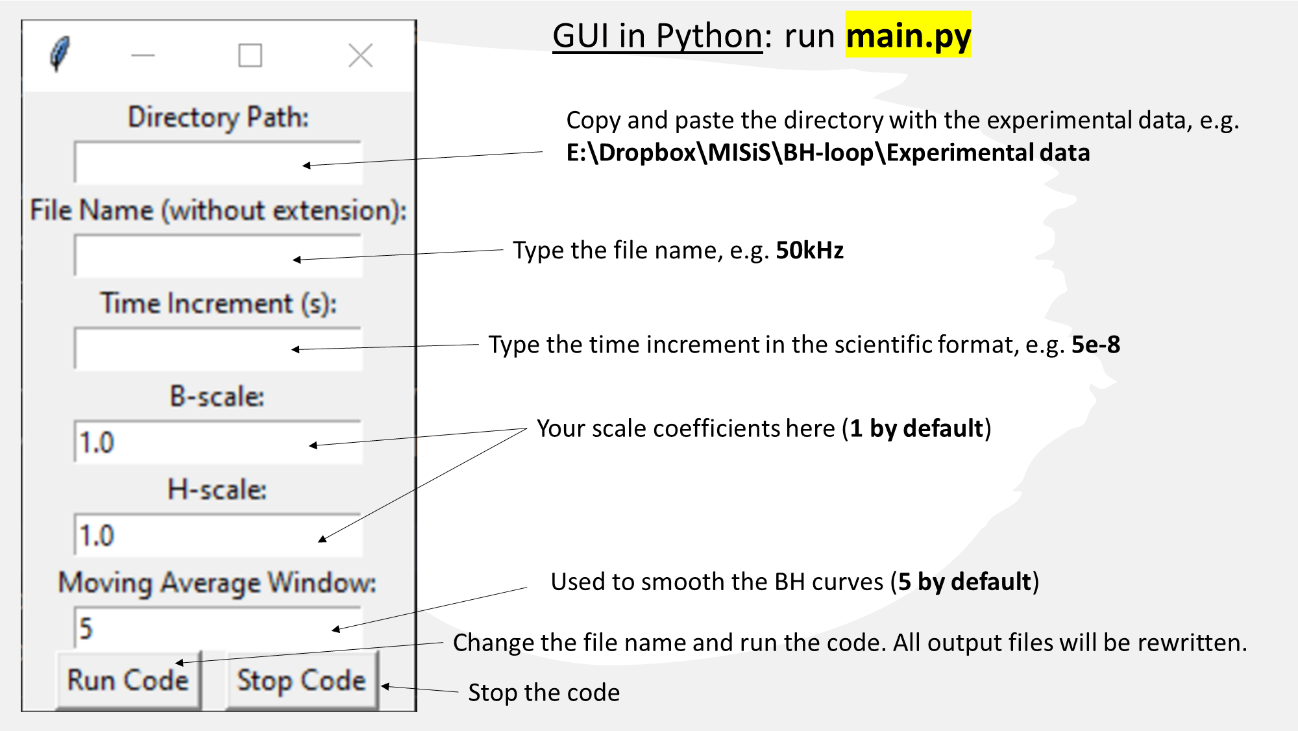
103.

104. # Start the GUI event loop

105. window.mainloop()

## BH-loop algorithm





Put your parameters (an example below) and run the code:

A screenshot of a computer

Description automatically generated

The initial figure generated by the code is displayed below. It will be saved in the PNG format. To proceed with the operation, you must close this figure.

A graph of sinusoid function

Description automatically generated

The following figure, generated by the code, represents a smoothed BH-loop and will be saved in the PNG format. The values on the x and y axes will attain genuine meaning once the B- and H-scales are established. To proceed with the operation, you must close this figure. Following that, you can enter a new file name and repeat the previous steps. Please be aware that all output files will be overwritten after each new execution. Therefore, to preserve these files for a specific experimental measurement, you must relocate them to a different folder.

A graph with a red line

Description automatically generated

Output files will be saved in the same folder as the experimental data:

A close-up of a computer code

Description automatically generated

Additionally, the code will print the sinusoid parameters in the program console window.

Parameters saved in the **signal\_parameters.txt** file:

1. File name and directory E:\Dropbox\MISiS\Digital BH-loop\Experimental data\50kHz.csv

2.

3. Time increment = 5e-08 s

4. Fitted sinusoid amplitude = 15.099995113879167 (your units)

5. Fitted sinusoid frequency = 49992.53693715317 Hz

6. Fitted sinusoid phase = 0.12758812005225068 rads

7. Fitted sinusoid phase = 7.310260795002432 degrees

8.

9. B-scale = 1.0

10. H-scale = 1.0

11.

12. Reference time t1 = 1.4596053025498542e-05 s

13. Reference time t2 = 2.4597545860890395e-05 s

14. Reference time t3 = 3.459903869628225e-05 s

15.

16. Reference index 1 = 291

17. Reference index 2 = 491

18. Reference index 3 = 691

19.

**BH.py (with GUI):**

1. #

2. # Digital BH-loop algorithm

3. # Project repository on GitHub: https://github.com/DYK-Team/Digital\_BH-loop\_algorithm

4. # 15/10/2023

5. #

6. # Authors:

7. # Ekaterina Nefedova, Dr. Mark Nemirovich, Dr. Nikolay Udanov, and Prof. Larissa Panina

8. # MISiS, Moscow, Russia, <https://en.misis.ru/>

9. #

10. # Dr. Dmitriy Makhnovskiy

11. # DYK+ team, United Kingdom, [www.dykteam.com](http://www.dykteam.com)

12. #

13.

14. import csv

15. import numpy as np

16. from scipy.optimize import curve\_fit

17. import matplotlib.pyplot as plt

18. import tkinter as tk

19.

20. # Default input parameter values

21. pi = np.pi # pi-constant 3.1415....

22. default\_B\_scale = 1.0 # This scale depends on the measurement units (V or mV) and the wire length and diameter

23. default\_H\_scale = 1.0 # This scale depends on the measurement units (V or mV) and the wire length and diameter

24. default\_window\_size = 5 # Moving Average Window. You can change from the GUI window.

25.

26. # Function to run the code with the entered parameters

27. def run\_code():

28. directory\_path = directory\_path\_entry.get() # Copy and paste the directory path to the GUI window

29. name = name\_entry.get() # Enter the file name without the CSV extension to the GUI window (e.g. 50kHz)

30. time\_increment = float(time\_increment\_entry.get()) # Enter the time increment (s) to the GUI window

31. B\_scale = float(B\_scale\_entry.get())

32. H\_scale = float(H\_scale\_entry.get())

33. window\_size = int(window\_size\_entry.get())

34.

35. # Full file name, including the directory path and the csv extension

36. file\_name = directory\_path + '\\' +name + '.csv'

37.

38. # Data from the CSV file (two columns without header)

39. data = np.genfromtxt(file\_name, delimiter=',')

40. response\_values = data[:, 0] # First column: response values which are proportional to the induction B

41. sin\_values = data[:, 1] # Second column: sinusoid values which are proportional to the scanning magnetic field H

42. N = len(sin\_values) # Number of points in each column

43.

44. # Time values based on the time increment

45. time = np.arange(0, N \* time\_increment, time\_increment)

46.

47. # Estimation for the sinusoid amplitude

48. A0 = np.max(sin\_values)

49.

50. # Scenarios for selecting sine wave vertices

51. scenario = 1 if sin\_values[0] >= 0 else 2

52.

53. # Skipping initial values until the sinusoid changes sign. After this, the search for vertices begins.

54. i = 0

55. while i <= N - 1 and ((scenario == 1 and sin\_values[i] >= 0) or (scenario == 2 and sin\_values[i] <= 0)):

56. i += 1

57. start = i # Start index

58.

59. # Searching for the indices of the first positive and negative sinusoid vertices

60.

61. # Set of indices near the positive vertex of the sinusoid

62. def pset(start, y\_values):

63. positive\_set = []

64. i = start

65. while i <= N - 1 and y\_values[i] >= 0:

66. if A0 \* 0.9 <= y\_values[i] <= A0:

67. positive\_set.append(i)

68. i += 1

69. stop = i

70. return np.array(positive\_set), stop

71.

72. # Set of indices near the negative vertex of the sinusoid

73. def nset(start, y\_values):

74. negative\_set = []

75. i = start

76. while i <= N - 1 and y\_values[i] <= 0:

77. if -A0 <= y\_values[i] <= -A0 \* 0.9:

78. negative\_set.append(i)

79. i += 1

80. stop = i

81. return np.array(negative\_set), stop

82.

83. if scenario == 1:

84. negative\_set, stop = nset(start, sin\_values)

85. positive\_set = pset(stop, sin\_values)[0]

86. elif scenario == 2:

87. positive\_set, stop = pset(start, sin\_values)

88. negative\_set = nset(stop, sin\_values)[0]

89.

90. # Average indices for the found sets of the negative and positive sinusoid vertices

91. positive\_set\_aver = float(sum(positive\_set) / len(positive\_set))

92. negative\_set\_aver = float(sum(negative\_set) / len(negative\_set))

93.

94. # Integer indices for the found negative and positive sinusoid vertices

95. pvertex = int(positive\_set\_aver)

96. nvertex = int(negative\_set\_aver)

97.

98. # Estimations of the period (T0) and frequency (f0) of the sinusoid

99. T0 = abs(positive\_set\_aver - negative\_set\_aver) \* time\_increment \* 2

100. f0 = 1 / T0

101.

102. # Improving A0: average amplitude of the sinusoid calculated from two vertices

103. A0 = (sin\_values[pvertex] - sin\_values[nvertex]) / 2.0

104.

105. # Estimation of the sinusoid phase (ph0) in radians

106. # Only positive phase will be used (anticlockwise rotation)

107. qT = int(abs(pvertex - nvertex) / 2) # Index difference for the quarter period

108. value = max(-1.0, min(sin\_values[0] / A0, 1.0)) # Normalising the zero-index value and clamping it to [-1, 1]

109. phase = np.arcsin(value) # Phase calculated from the arcsin function in the range [-pi/2, pi/2]

110. if scenario == 1: # First scenario sin\_values[0] >= 0

111. if sin\_values[qT] >= 0:

112. ph0 = phase # First quarter

113. else:

114. ph0 = pi - phase # Second quarter

115. else: # Second scenario sin\_values[0] <= 0

116. if sin\_values[qT] <= 0:

117. ph0 = pi - phase # Third quarter

118. else:

119. ph0 = 2.0 \* pi + phase # Fourth quarter

120.

121. print('Estimated amplitude = ', A0)

122. print('Estimated frequency = ', f0, ' Hz')

123. print('Estimated phase = ', ph0, ' rads')

124. print('Estimated phase = ', np.degrees(ph0), ' degrees')

125.

126. # Sinusoidal function used in the fitting

127. def sinusoid(t, A, f, phase):

128. return A \* np.sin(2 \* pi \* f \* t + phase)

129.

130. # Fitting the data (second column) to the sinusoid

131. params, covariance = curve\_fit(sinusoid, time, sin\_values, p0=[A0, f0, ph0]) # [A0, f0, ph0] - initial values

132.

133. # Extracted fitting parameters

134. A\_fit, f\_fit, ph\_fit = params

135.

136. # Fitted sinusoid curve

137. sinusoid\_fit = sinusoid(time, A\_fit, f\_fit, ph\_fit)

138. ph\_degrees = np.degrees(ph\_fit)

139.

140. print('')

141. print('Fitted amplitude = ', A\_fit)

142. print('Fitted frequency = ', f\_fit, ' Hz')

143. print('Fitted phase = ', ph\_fit, ' rads')

144. print('Fitted phase = ', ph\_degrees, ' degrees')

145.

146. # Calculation of the reference time points t123 used in the numerical integration

147. if scenario == 1: # First scenario sin\_values[0] >= 0

148. t1 = (3.0 \* pi / 2.0 - ph\_fit) / (2.0 \* pi \* f\_fit)

149. t2 = (5.0 \* pi / 2.0 - ph\_fit) / (2.0 \* pi \* f\_fit)

150. else: # Second scenario sin\_values[0] <= 0

151. t1 = (5.0 \* pi / 2.0 - ph\_fit) / (2.0 \* pi \* f\_fit)

152. t2 = (7.0 \* pi / 2.0 - ph\_fit) / (2.0 \* pi \* f\_fit)

153. t3 = t2 + 0.5 / f\_fit

154.

155. print('')

156. print('Reference time t1 = ', t1, ' s')

157. print('Reference time t2 = ', t2, ' s')

158. print('Reference time t3 = ', t3, ' s')

159.

160. # Indexes corresponding to the reference time moments

161. refindex1 = int(t1 / time\_increment)

162. refindex2 = int(t2 / time\_increment)

163. refindex3 = int(t3 / time\_increment)

164.

165. print('')

166. print('Reference index 1 = ', refindex1)

167. print('Reference index 2 = ', refindex2)

168. print('Reference index 3 = ', refindex3)

169.

170. # Writing the parameters to the txt file

171. with open(directory\_path + '\\' + 'signal\_parameters.txt', 'w') as file:

172. file.write('\n')

173. file.write('File name and directory {}\n'.format(file\_name))

174. file.write('\n')

175. file.write('Time increment = {} s\n'.format(time\_increment))

176. file.write('Fitted sinusoid amplitude = {} (your units)\n'.format(A\_fit))

177. file.write('Fitted sinusoid frequency = {} Hz\n'.format(f\_fit))

178. file.write('Fitted sinusoid phase = {} rads\n'.format(ph\_fit))

179. file.write('Fitted sinusoid phase = {} degrees\n'.format(ph\_degrees))

180. file.write('\n')

181. file.write('B-scale = {} \n'.format(B\_scale))

182. file.write('H-scale = {} \n'.format(H\_scale))

183. file.write('\n')

184. file.write('Reference time t1 = {} s \n'.format(t1))

185. file.write('Reference time t2 = {} s \n'.format(t2))

186. file.write('Reference time t3 = {} s \n'.format(t3))

187. file.write('\n')

188. file.write('Reference index 1 = {} \n'.format(refindex1))

189. file.write('Reference index 2 = {} \n'.format(refindex2))

190. file.write('Reference index 3 = {} \n'.format(refindex3))

191.

192. # Plot the original data and the fitted curve

193. plt.figure(figsize=(10, 6))

194. plt.scatter(time, sin\_values, label='Original Data', color='blue', marker='o')

195. plt.plot(time, sinusoid\_fit, label='Fitted Sinusoid', color='red')

196.

197. # Add vertical lines at t1, t2, and t3

198. plt.axvline(x=t1, color='green', linestyle='--', label='t1')

199. plt.axvline(x=t2, color='purple', linestyle='--', label='t2')

200. plt.axvline(x=t3, color='orange', linestyle='--', label='t3')

201.

202. plt.xlabel('Time')

203. plt.ylabel('Amplitude')

204. plt.title('Sinusoidal Fit with Reference Points t1, t2, and t3')

205. plt.legend()

206. plt.grid(True)

207.

208. # Saving the plot as an image

209. plt.savefig(directory\_path + '\\' + 'sinusoid\_fitting\_reference\_points.png')

210.

211. plt.show()

212.

213. # Forward integration of the voltage response between the reference indexes 1 and 2

214. B\_forward = []

215. H\_forward = []

216. integral\_value = 0.0

217. for i in range(refindex1, refindex2):

218. H\_forward.append(sin\_values[i])

219. for i in range(refindex1, i):

220. integral\_value += 0.5 \* (response\_values[i] + response\_values[i + 1]) \* time\_increment # Trapezoid method

221. B\_forward.append(integral\_value)

222.

223. # Reverse integration of the voltage response between the reference indexes 2 and 3

224. B\_reverse = []

225. H\_reverse = []

226. integral\_value = 0.0

227. for i in range(refindex2, refindex3):

228. H\_reverse.append(sin\_values[i])

229. for i in range(refindex2, i):

230. integral\_value += 0.5 \* (response\_values[i] + response\_values[i + 1]) \* time\_increment # Trapezoid method

231. B\_reverse.append(integral\_value)

232.

233. # Rescaling the magnetic induction B and the field H from the data values

234. B\_forward = np.array(B\_forward) \* B\_scale

235. H\_forward = -np.array(H\_forward) \* H\_scale

236. lf = len(B\_forward) # Number of points in the forward BH curve

237. B\_reverse = np.array(B\_reverse) \* B\_scale

238. H\_reverse = -np.array(H\_reverse) \* H\_scale

239. lr = len(B\_reverse) # Number of points in the reverse BH curve

240.

241. # Defining the concavity con\_forward of the forward BH curve

242. # B = af + bf \* H is the straight line between the forward BH curve ends

243. bf = (B\_forward[lf - 1] - B\_forward[0]) / (H\_forward[lf - 1] - H\_forward[0])

244. af = (B\_forward[0] \* H\_forward[lf - 1] - B\_forward[lf - 1] \* H\_forward[0]) / (H\_forward[lf - 1] - H\_forward[0])

245. # Direction of the concavity

246. if (af + bf \* (H\_forward[lf - 1] - H\_forward[0]) / 2) >= B\_forward[int(lf / 2)]:

247. con\_forward = 'down'

248. else:

249. con\_forward = 'up'

250.

251. # Defining the concavity con\_reverse of the reverse BH curve

252. # B = ar + br \* H is the straight line between the reverse BH curve ends

253. br = (B\_reverse[lr - 1] - B\_reverse[0]) / (H\_reverse[lr - 1] - H\_reverse[0])

254. ar = (B\_reverse[0] \* H\_reverse[lr - 1] - B\_reverse[lr - 1] \* H\_reverse[0]) / (H\_reverse[lr - 1] - H\_reverse[0])

255. # Direction of the concavity

256. if (ar + br \* (H\_reverse[lr - 1] - H\_reverse[0]) / 2) >= B\_reverse[int(lr / 2)]:

257. con\_reverse = 'down'

258. else:

259. con\_reverse = 'up'

260.

261. # Vertical shifts of the BH curves

262. if con\_forward == 'up':

263. B\_forward = B\_forward + abs(B\_forward[0] - B\_forward[lf - 1]) / 2

264. else:

265. B\_forward = B\_forward - abs(B\_forward[0] - B\_forward[lf - 1]) / 2

266.

267. if con\_reverse == 'up':

268. B\_reverse = B\_reverse + abs(B\_reverse[0] - B\_reverse[lr - 1]) / 2

269. else:

270. B\_reverse = B\_reverse - abs(B\_reverse[0] - B\_reverse[lr - 1]) / 2

271.

272. # Function for computing the moving average

273. def moving\_average(data, window\_size):

274. cumsum = np.cumsum(data)

275. cumsum[window\_size:] = cumsum[window\_size:] - cumsum[:-window\_size]

276. return cumsum[window\_size - 1:] / window\_size

277.

278. # Smoothing the data using moving averages

279. B\_forward\_smoothed = moving\_average(B\_forward, window\_size)

280. H\_forward\_smoothed = moving\_average(H\_forward, window\_size)

281. B\_reverse\_smoothed = moving\_average(B\_reverse, window\_size)

282. H\_reverse\_smoothed = moving\_average(H\_reverse, window\_size)

283.

284. # Creating the graph of smoothed curves

285. plt.figure(figsize=(10, 6))

286. plt.plot(H\_forward\_smoothed, B\_forward\_smoothed, label='Smoothed B\_forward vs. H\_forward', color='blue')

287. plt.plot(H\_reverse\_smoothed, B\_reverse\_smoothed, label='Smoothed B\_reverse vs. H\_reverse', color='red')

288. plt.xlabel('H (A/m)')

289. plt.ylabel('B (T)')

290. plt.title('Smoothed Magnetic Hysteresis Loop')

291. plt.legend()

292. plt.grid(True)

293.

294. # Saving the data and smoothed curves to a CSV file

295. data = np.column\_stack((H\_forward\_smoothed, B\_forward\_smoothed, H\_reverse\_smoothed, B\_reverse\_smoothed))

296. header = ['H\_forward (A/m)', 'B\_forward\_smoothed (T)', 'H\_reverse (A/m)', 'B\_reverse\_smoothed (T)']

297.

298. with open(directory\_path + '\\' + 'smoothed\_hysteresis\_data.csv', 'w', newline='') as csv\_file:

299. writer = csv.writer(csv\_file)

300. writer.writerow(header)

301. writer.writerows(data)

302.

303. # Saving the plot as an image

304. plt.savefig(directory\_path + '\\' + 'smoothed\_hysteresis\_plot.png')

305.

306. plt.show()

307.

308. # Create a function to stop the code execution

309. def stop\_code():

310. quit()

311.

312. # Main GUI window

313. root = tk.Tk()

314. root.title("Input Parameters")

315.

316. # Labels and entry fields for input parameters

317. directory\_path\_label = tk.Label(root, text="Directory Path:")

318. directory\_path\_label.pack()

319. directory\_path\_entry = tk.Entry(root)

320. directory\_path\_entry.pack()

321.

322. name\_label = tk.Label(root, text="File Name (without extension):")

323. name\_label.pack()

324. name\_entry = tk.Entry(root)

325. name\_entry.pack()

326.

327. time\_increment\_label = tk.Label(root, text="Time Increment (s):")

328. time\_increment\_label.pack()

329. time\_increment\_entry = tk.Entry(root)

330. time\_increment\_entry.pack()

331.

332. B\_scale\_label = tk.Label(root, text="B-scale:")

333. B\_scale\_label.pack()

334. B\_scale\_entry = tk.Entry(root)

335. B\_scale\_entry.insert(0, default\_B\_scale) # Default value

336. B\_scale\_entry.pack()

337.

338. H\_scale\_label = tk.Label(root, text="H-scale:")

339. H\_scale\_label.pack()

340. H\_scale\_entry = tk.Entry(root)

341. H\_scale\_entry.insert(0, default\_H\_scale) # Default value

342. H\_scale\_entry.pack()

343.

344. window\_size\_label = tk.Label(root, text="Moving Average Window:")

345. window\_size\_label.pack()

346. window\_size\_entry = tk.Entry(root)

347. window\_size\_entry.insert(0, default\_window\_size) # Default value

348. window\_size\_entry.pack()

349.

350. # Frame to hold the buttons in one row

351. button\_frame = tk.Frame(root)

352. button\_frame.pack()

353.

354. # "Run Code" button with some padding to the right

355. run\_button = tk.Button(button\_frame, text="Run Code", command=run\_code)

356. run\_button.pack(side=tk.LEFT, padx=5) # Adjust the padx value as needed

357.

358. # "Stop Code" button with some padding to the left

359. stop\_button = tk.Button(button\_frame, text="Stop Code", command=stop\_code)

360. stop\_button.pack(side=tk.LEFT, padx=5) # Adjust the padx value as needed

361.

362. # Main event loop

363. root.mainloop()

364.