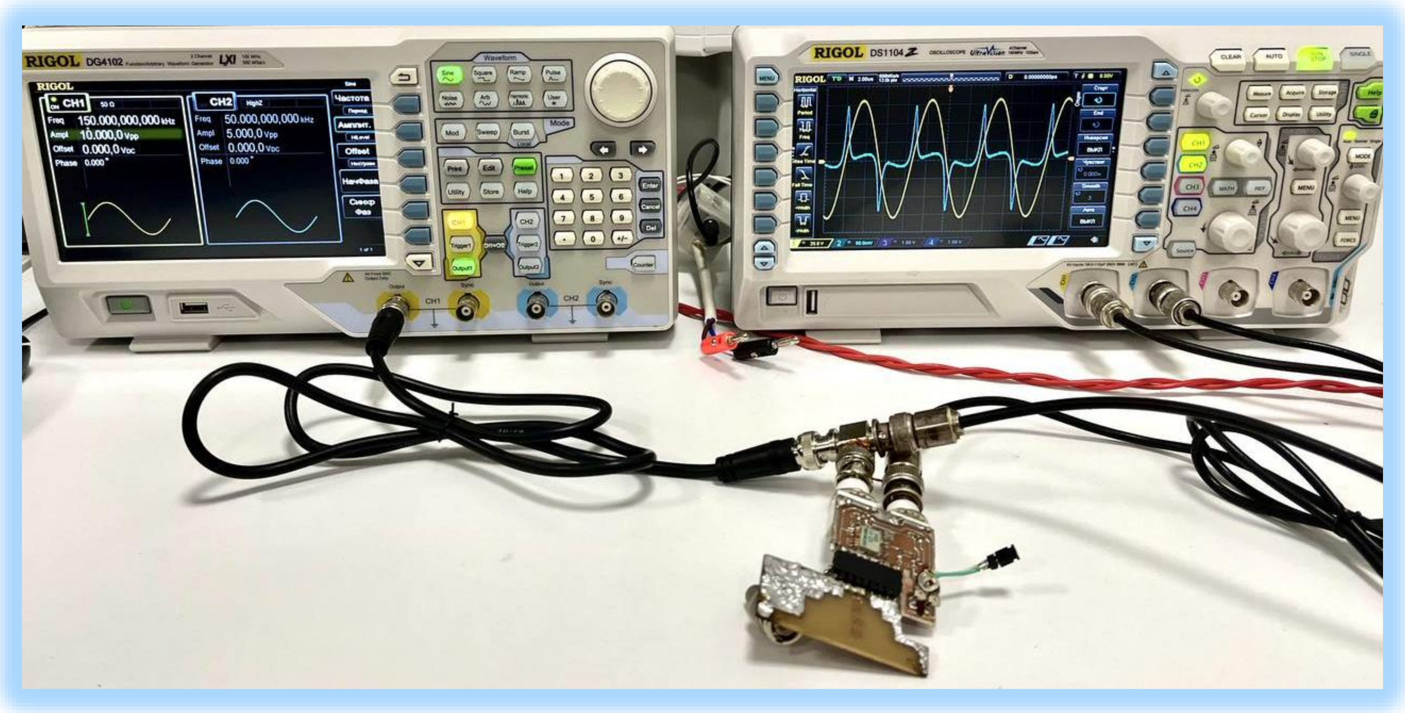
MISiS, https://en.misis.ru/, Moscow, Russia DYK+ team, www.dykteam.com, United Kingdom 16/10/2023

Digital BH-loop measurement setup and algorithm



Contents

Measurement setup 2
Selection of the scales for H and B 5
Appendix: algorithms in Python 8
Wheatstone calculator 8
BH-loop algorithm 11

Project package: 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 (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 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.

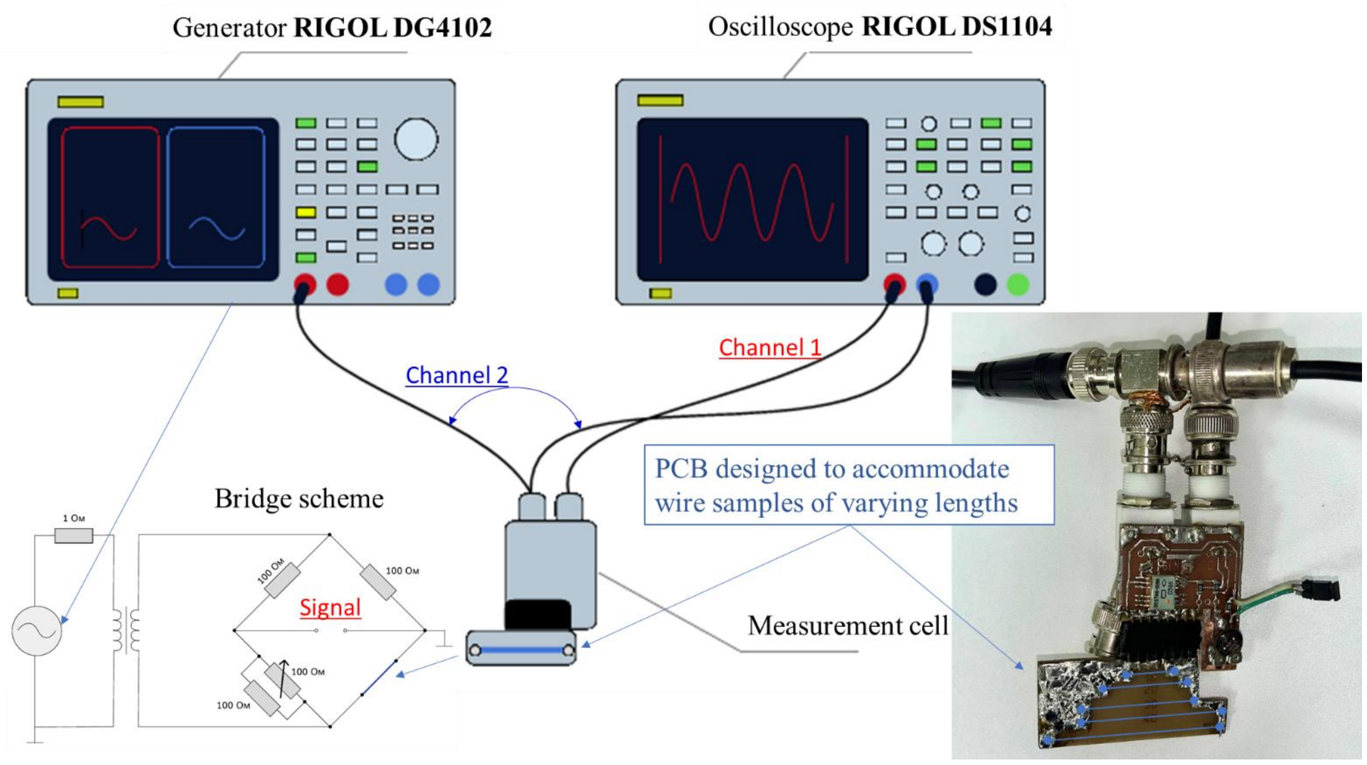


Fig. 1. Measurement setup.

Let us consider the classical Wheatstone bridge circuit 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 R_0 – in or case this is the resistance of the transformer secondary coil. The balancing, when the output voltage V_{out} is zero, requires the following condition:

$$R_2 = \frac{R_w \times R_1}{R_3} \tag{1}$$

where R_2 is the value of the tuning resistor and R_w is the microwire DC resistance. The general equation for the potential difference across R_w is as follows:

$$V_w = V_0 \times \frac{(R_1+R_2)R_w}{R_0(R_1+R_2+R_3+R_w)+(R_1+R_2)(R_3+R_w)} \tag{2}$$

We require this transfer function to establish a scale for the magnetizing field H. We have developed a simple Wheatstone calculator in Python for calculating transfer functions in a balanced bridge circuit (see Appendix). Additionally, you can utilize the circuit in the LTspice simulator (saved in the "Wheatstone_Calculator_LTspice_Vw_V0" project folder), with its interface shown in Fig. 3.

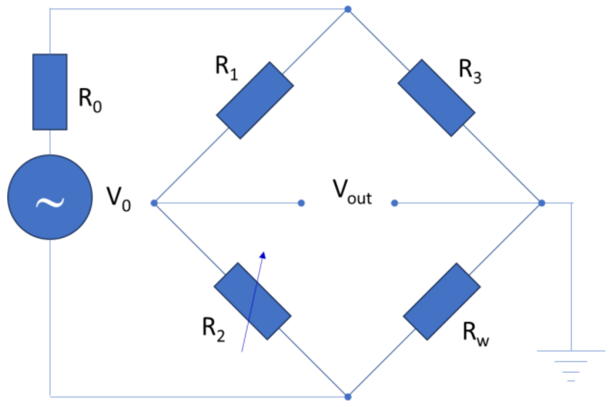
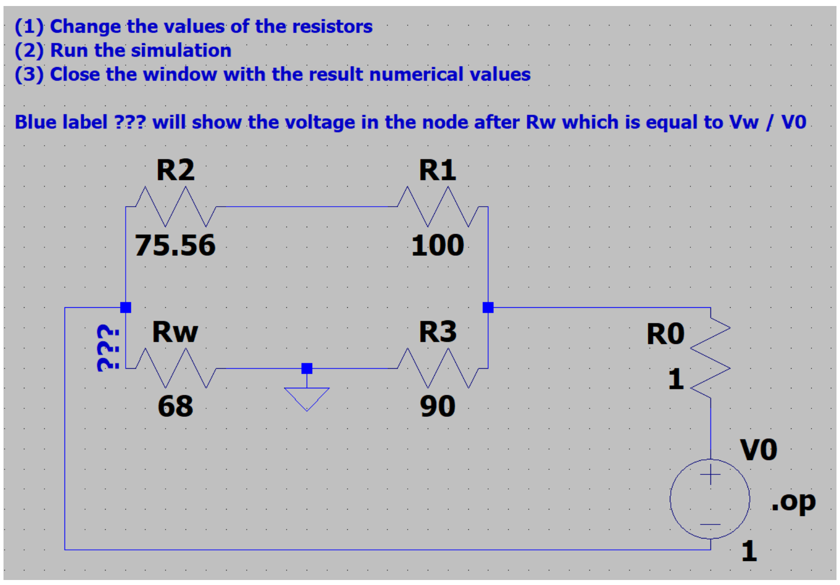


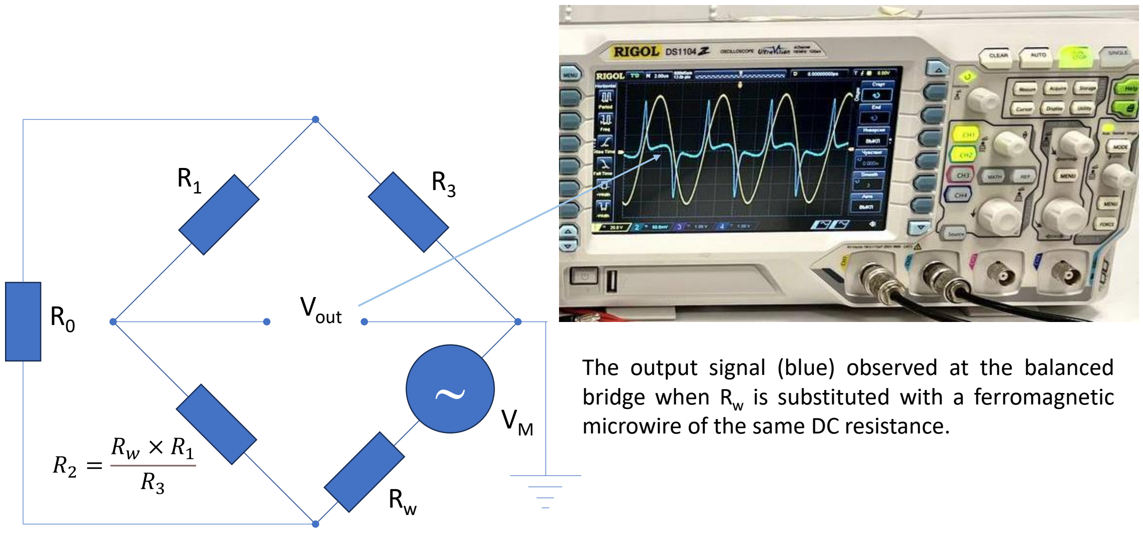
Fig. 2. Wheatstone bridge circuit.


(1) Change the values of the resistors
(2) Run the simulation
(3) Close the window with the result numerical values

Blue label ??? will show the voltage in the node after Rw which is equal to Vw / V0

Fig. 3. LTspice circuit for calculating the $V_0 \rightarrow V_w$ transfer function. $V_0 = 1$.

Balancing the bridge, when the resistor R_w is substituted with a ferromagnetic microwire of the same DC resistance, results in the detection of an additional electromotive force (emf) V_M. 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 V_0 but featuring the additional source V_M, is illustrated in Fig. 4.



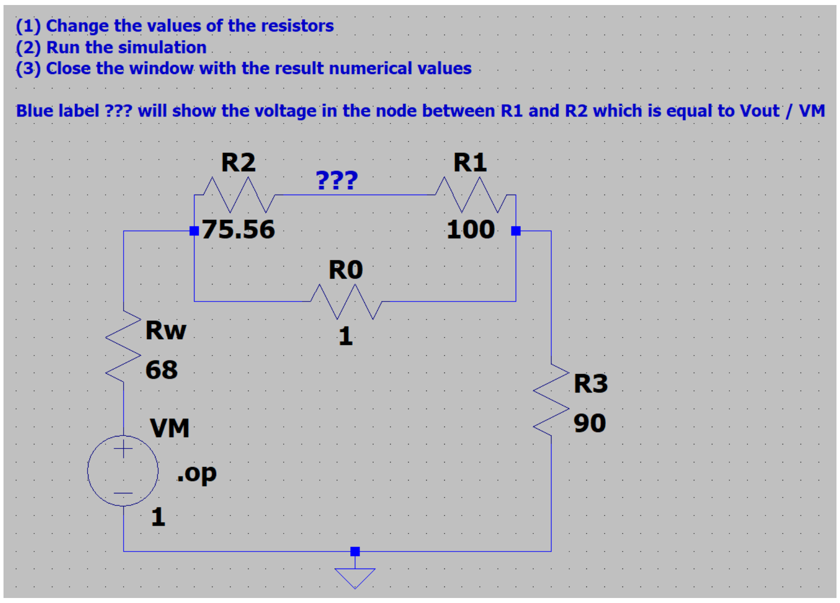
The output signal (blue) observed at the balanced bridge when R_w is substituted with a ferromagnetic microwire of the same DC resistance.

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

While V_{out} will be the measured parameter, V_M is essential for calculating the BH-loops. Consequently, we have established the $V_{out} \rightarrow V_M$ transfer function (see Wheatstone calculator in Appendix):

$$V_M = \frac{R_3 + R_w}{R_3} \times V_{out} \tag{3}$$

Additionally, you can utilize the circuit in the LTspice simulator (saved in the "Wheatstone_Calculator_LTspice_VM_Vout" project folder), with its interface shown in Fig. 5.


(1) Change the values of the resistors
(2) Run the simulation
(3) Close the window with the result numerical values

Blue label ??? will show the voltage in the node between R1 and R2 which is equal to Vout / VM

Fig. 5. LTspice circuit for calculating the $V_M \rightarrow V_{out}$ inverse transfer function. $V_M = 1$.

Selection of the scales for H and B

The following rationale aims to propose a reasonable selection of scales for the magnetizing field H and magnetic induction B. Let us examine Maxwell's equation for Ampere's circuital law in its integral form (SI units):

$$\oint_{\partial\Sigma}(\vec{\boldsymbol{H}},\vec{\boldsymbol{\tau}})dl = \oiint_{\Sigma}(\vec{\boldsymbol{J}}_f,\vec{\boldsymbol{n}})ds + \frac{d}{dt}\oiint_{\Sigma}(\vec{\boldsymbol{D}},\vec{\boldsymbol{n}})ds \qquad (4)$$

where $\vec{\boldsymbol{J}}_f$ represents the vector of current density for free charges across a surface Σ, $\partial\Sigma$ is the contour of Σ's edge traversed in the positive direction (with the contour being on the left), $(\vec{\boldsymbol{H}},\vec{\boldsymbol{\tau}})$ denotes the scalar product of the magnetizing field $\vec{\boldsymbol{H}}$ and the unit vector $\vec{\boldsymbol{\tau}}$, which is tangent in the positive direction to $\partial\Sigma$ at each integration point. Meanwhile, $\vec{\boldsymbol{n}}$ is a unit vector normal to the surface Σ at each integration point, and its direction is determined according to the gimlet rule while rotating in the positive direction along the traversal of $\partial\Sigma$. Lastly, $(\vec{\boldsymbol{D}},\vec{\boldsymbol{n}})$ represents the scalar product of the displacement vector $\vec{\boldsymbol{D}}$ and $\vec{\boldsymbol{n}}$.

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 πr^2 with a variable radius $0 < r < a$ (a is the microwire radius) as the integration surface Σ, we can represent the surface integral over the current density as follows:

$$\oiint_{\Sigma}(\vec{\boldsymbol{J}}_f,\vec{\boldsymbol{n}})ds = I \times \frac{r^2}{a^2} \qquad (5)$$

where I 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:

$$\oint_{\partial\Sigma}(\vec{\boldsymbol{H}},\vec{\boldsymbol{\tau}})dl = H_\varphi(r)2\pi r \qquad (6)$$

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

$$H_\varphi(r) = I \times \frac{r}{2\pi a^2} \qquad (7)$$

Unlike the measurement scheme for longitudinal hysteresis loops, this field exhibits significant inhomogeneity along the radius. Hence, the average value $I/(4\pi a)$ 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):

$$H_\varphi[\mathrm{A/m}] = \frac{V_w[V]}{4\pi a[m]\times R_w[\Omega]} \qquad (8)$$

The dimensions are indicated within square brackets. Here, V_w should be calculated from V_0, which is measured on the second channel of the oscilloscope (reference sinusoid), following Eq. (2). Note that $R_w[\Omega]$ represents the DC resistance of the microwire.

Now, we shall establish a measurement scale for magnetic induction by utilizing the Faraday-Maxwell equation in its integral form (SI units):

$$\oint_{\partial\Sigma}(\vec{\mathbf{E}},\vec{\boldsymbol{\tau}})dl = -\frac{d}{dt}\oiint_{\Sigma}(\vec{\mathbf{B}},\vec{\boldsymbol{n}})ds = -\frac{d}{dt}\oiint_{\Sigma}(\mu_0\vec{\mathbf{H}}+\mu_0\vec{\mathbf{M}},\vec{\boldsymbol{n}})ds \quad (9)$$

where $\vec{\mathbf{E}}$ is the vector of electrical field, $\vec{\mathbf{M}}$ is the vector of magnetization, and μ_0 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 $\oiint_{\Sigma}(\mu_0\vec{\mathbf{H}},\vec{\boldsymbol{n}})ds$ contributing to the emf is compensated by the bridge circuit. By performing contour integration from the left, we obtain:

$$2E(t)l = 2V_M(t) = 2\times\frac{R_3+R_w}{R_3}\times V_{out}(t) = -\frac{d}{dt}\oiint_{\Sigma_1}(\mu_0\vec{\mathbf{M}},\vec{\boldsymbol{n}})ds - \frac{d}{dt}\oiint_{\Sigma_2}(\mu_0\vec{\mathbf{M}},\vec{\boldsymbol{n}})ds \quad (10)$$

where $V_M(t)$ is the voltage induced by the magnetization reversal, and $V_{out}(t)$ is the voltage measured on the first channel of the oscilloscope (see Eq. (3)). Both areas $\Sigma_{1,2} = a\times l$ (see Fig. 6), where a and l are the wire radius and length respectively. The surface integrals on the right-hand side of Eq. (10), representing the magnetic induction flux $\Phi(t)$, have dimensions in Webers (Wb). To obtain the result in Teslas (T), both sides should be normalized by the integration area $a\times l$:

$$\frac{d}{dt}\frac{\Phi(t)}{a\times l} = \frac{R_3+R_w}{R_3\times a\times l}\times V_{out}(t) \quad (11)$$

Hence, the magnetic induction $B(t)$ averaged over half of the longitudinal section ($a\times l$) can be reconstructed through time integration:

$$B(t)[T] = \frac{(R_3+R_W)[\Omega]}{R_3[\Omega]\times a[m]\times l[m]}\times\int V_{out}(t)dt \quad (12)$$

The dimensions are indicated within square brackets. The voltage output $V_{out}(t)$ 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.

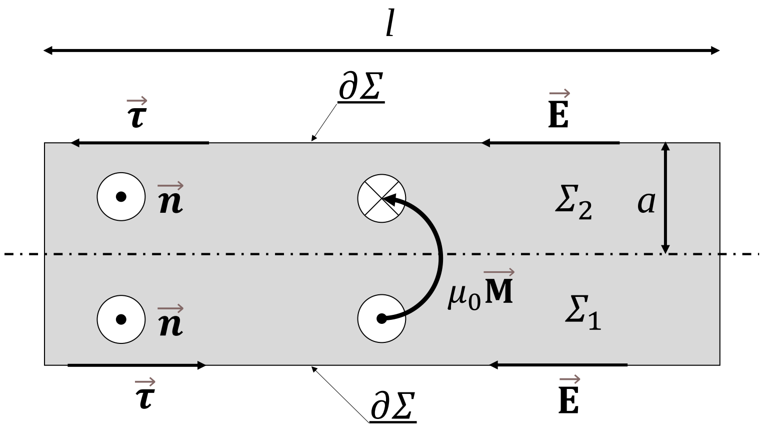
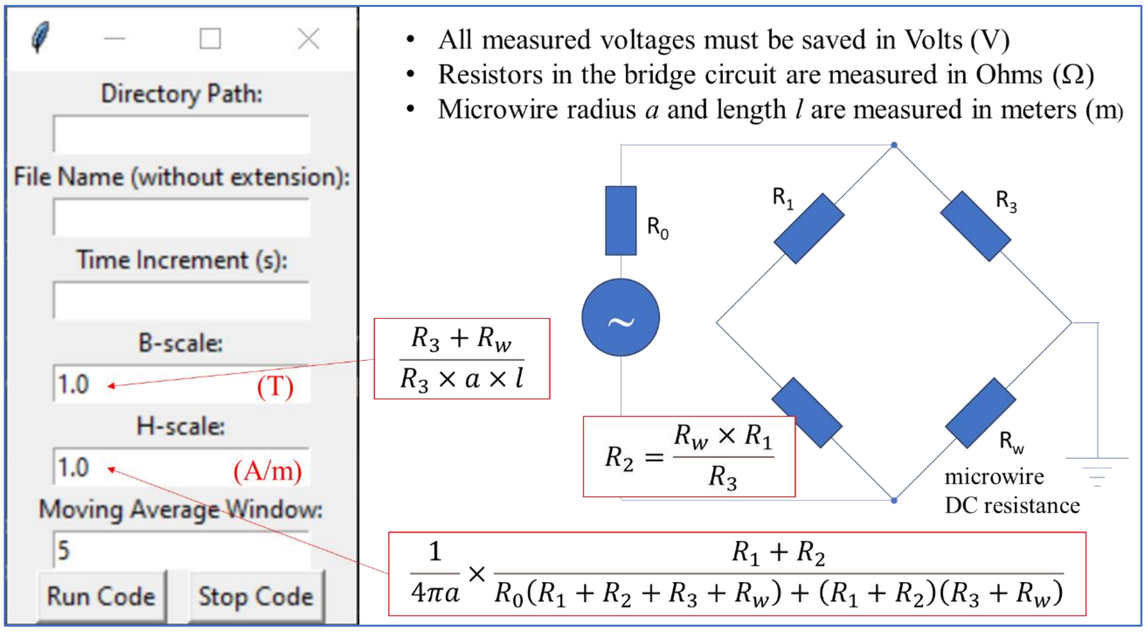


Fig. 6. Longitudinal section of the microwire representing the integration geometry for Eq. (9).

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



Directory Path:

File Name (without extension):

Time Increment (s):

B-scale: 1.0 (T)

H-scale: 1.0 (A/m)

Moving Average Window: 5

- All measured voltages must be saved in Volts (V)
- Resistors in the bridge circuit are measured in Ohms (Ω)
- Microwire radius a and length l are measured in meters (m)

$$\frac{R_3 + R_w}{R_3 \times a \times l}$$

$$R_2 = \frac{R_w \times R_1}{R_3}$$

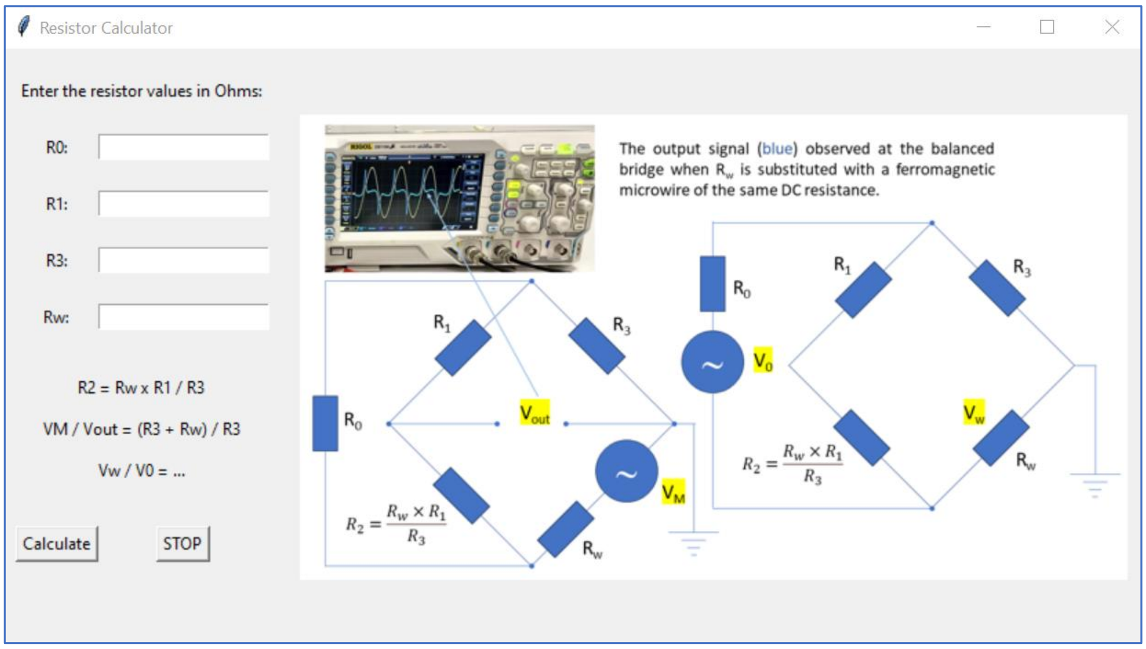
$$\frac{1}{4\pi a} \times \frac{R_1 + R_2}{R_0(R_1 + R_2 + R_3 + R_w) + (R_1 + R_2)(R_3 + R_w)}$$

Fig. 7. Rules for calculating the dimensional coefficients.

Appendix: algorithms in Python

Wheatstone calculator

This GUI algorithm calculates two transfer functions: $V_0 \rightarrow V_w$ and $V_{out} \rightarrow V_M$, which are necessary to establish scales for the magnetizing field H and magnetic induction B.



Resistor Calculator

Enter the resistor values in Ohms:

R0:

R1:

R3:

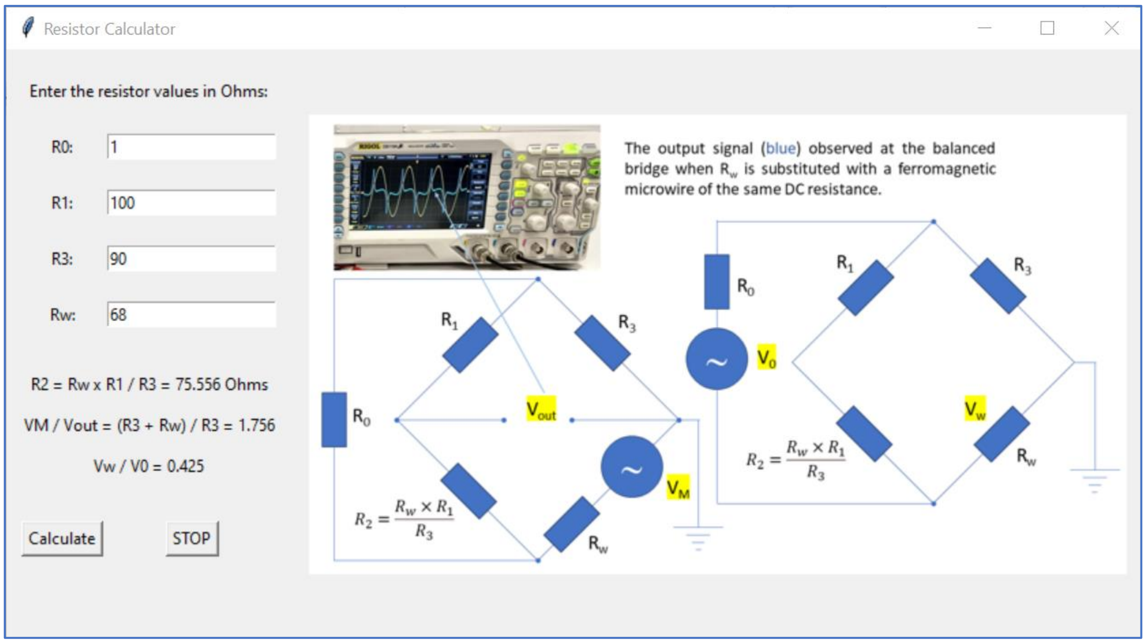
Rw:

R2 = Rw x R1 / R3

VM / Vout = (R3 + Rw) / R3

Vw / V0 = ...

The output signal (blue) observed at the balanced bridge when R_w is substituted with a ferromagnetic microwire of the same DC resistance.



Resistor Calculator

Enter the resistor values in Ohms:

R0: 1

R1: 100

R3: 90

Rw: 68

R2 = Rw x R1 / R3 = 75.556 Ohms

VM / Vout = (R3 + Rw) / R3 = 1.756

Vw / V0 = 0.425

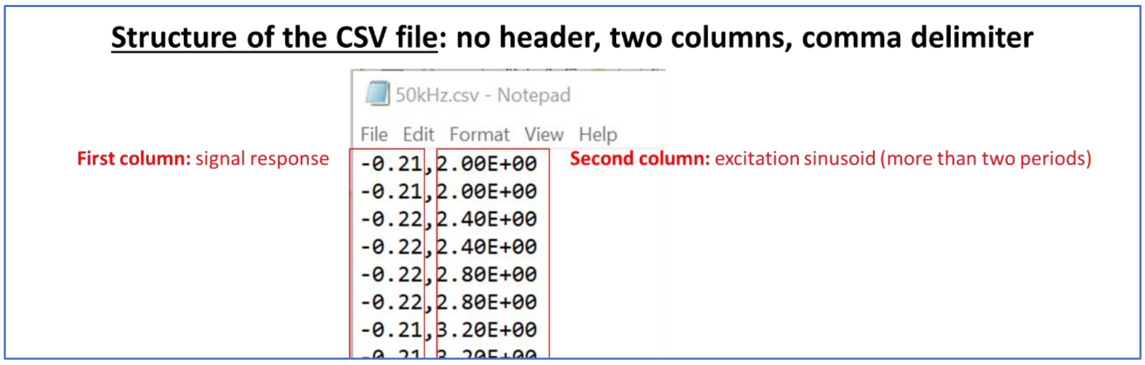
The output signal (blue) observed at the balanced bridge when R_w is substituted with a ferromagnetic microwire of the same DC resistance.

Wheatstone.py (with GUI):

```python
#
# Wheatstone calculator
# Project repository on GitHub: https://github.com/DYK-Team/Digital_BH-loop_algorithm
# 16/10/2023
#
# Authors:
# Ekaterina Nefedova, Dr. Mark Nemirovich, Dr. Nikolay Udanov, and Prof. Larissa Panina
# MISiS, Moscow, Russia, https://en.misis.ru/
#
# Dr. Dmitriy Makhnovskiy
# DYK+ team, United Kingdom, www.dykteam.com
#

import tkinter as tk
from PIL import Image, ImageTk

def calculate():
    R0 = float(R0_entry.get())
    R1 = float(R1_entry.get())
    R3 = float(R3_entry.get())
    Rw = float(Rw_entry.get())

    # Calculate R2 that provides the balance of the bridge circuit
    R2 = R1 * Rw / R3

    # Calculate VM / Vout
    VM_Vout = (R3 + Rw) / R3

    # Calculate Vw / V0
    Vw_V0 = ((R1 + R2) * Rw) / (R0 * (R1 + R2 + R3 + Rw) + (R1 + R2) * (R3 + Rw))

    # Update the equation label with the calculated R2 value and the value for VM / Vout and Vw / V0
    equation_label.config(text=f"R2 = Rw x R1 / R3 = {R2:.3f} Ohms\n\nVM / Vout = (R3 + Rw) / R3 = 
{VM_Vout:.3f}"
                               f"\n\nVw / V0 = {Vw_V0:.3f}")

# Function to stop the calculation and close the program
def stop_calculation():
    window.quit()

# Create the main window
window = tk.Tk()
window.title("Resistor Calculator")

# Create a frame for the left side
left_frame = tk.Frame(window)
left_frame.grid(row=0, column=0, padx=10, pady=10)  # Increased space using pady

# Create a frame for the right side (picture)
right_frame = tk.Frame(window)
right_frame.grid(row=0, column=1, padx=10, pady=10)  # Increased space using pady

# Add a message for entering resistor values
message_label = tk.Label(left_frame, text="Enter the resistor values in Ohms:")
message_label.grid(row=0, columnspan=2, pady=10)  # Increased space using pady

# Create labels and entry fields for resistor values with increased space
R0_label = tk.Label(left_frame, text="R0:")
R0_label.grid(row=1, column=0, pady=10)  # Increased space using pady
R0_entry = tk.Entry(left_frame)
R0_entry.grid(row=1, column=1, pady=10)  # Increased space using pady

R1_label = tk.Label(left_frame, text="R1:")
R1_label.grid(row=2, column=0, pady=10)  # Increased space using pady
R1_entry = tk.Entry(left_frame)
R1_entry.grid(row=2, column=1, pady=10)  # Increased space using pady

R3_label = tk.Label(left_frame, text="R3:")
R3_label.grid(row=3, column=0, pady=10)  # Increased space using pady
R3_entry = tk.Entry(left_frame)
```

```
R3_entry.grid(row=3, column=1, pady=10)  # Increased space using pady

Rw_label = tk.Label(left_frame, text="Rw:")
Rw_label.grid(row=4, column=0, pady=10)  # Increased space using pady
Rw_entry = tk.Entry(left_frame)
Rw_entry.grid(row=4, column=1, pady=10)  # Increased space using pady

# Create a label to display the R2 and Vw / Vout equations with increased space
equation_label = tk.Label(left_frame, text="R2 = Rw x R1 / R3\n\nVM / Vout = (R3 + Rw) / R3\n\nVw / V0 = ...")
equation_label.grid(row=5, columnspan=2, pady=20)  # Increased space using pady

# Create buttons to trigger the calculation and stop the calculation with increased space
calculate_button = tk.Button(left_frame, text="Calculate", command=calculate)
calculate_button.grid(row=6, column=0, pady=10)  # Increased space using pady

stop_button = tk.Button(left_frame, text="STOP", command=stop_calculation)
stop_button.grid(row=6, column=1, pady=10)  # Increased space using pady

# Create a label to display the result
result_label = tk.Label(left_frame, text="")
result_label.grid(row=7, columnspan=2, pady=10)  # Increased space using pady

# Open and resize the PNG image using Pillow without any filters
image = Image.open("Circuit.png")
new_width = 600  # Adjust the desired width for a larger picture
aspect_ratio = float(new_width) / float(image.width)
new_height = int(image.height * aspect_ratio)
image = image.resize((new_width, new_height))

photo = ImageTk.PhotoImage(image)
image_label = tk.Label(right_frame, image=photo)
image_label.photo = photo  # To prevent the image from being garbage collected
image_label.grid(row=0, column=0)

# Start the GUI event loop
window.mainloop()
```

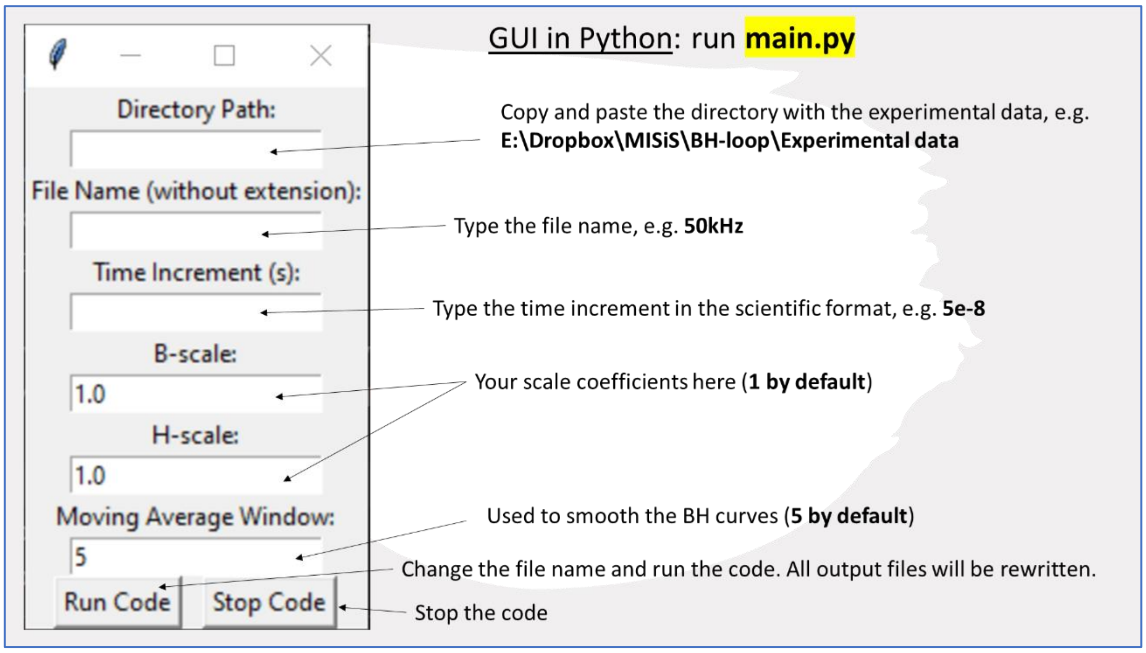
BH-loop algorithm


Structure of the CSV file: no header, two columns, comma delimiter

First column: signal response

Second column: excitation sinusoid (more than two periods)

```
-0.21,2.00E+00
-0.21,2.00E+00
-0.22,2.40E+00
-0.22,2.40E+00
-0.22,2.80E+00
-0.22,2.80E+00
-0.21,3.20E+00
```


GUI in Python: run **main.py**

Copy and paste the directory with the experimental data, e.g. **E:\Dropbox\MISiS\BH-loop\Experimental data**

Type the file name, e.g. **50kHz**

Type the time increment in the scientific format, e.g. **5e-8**

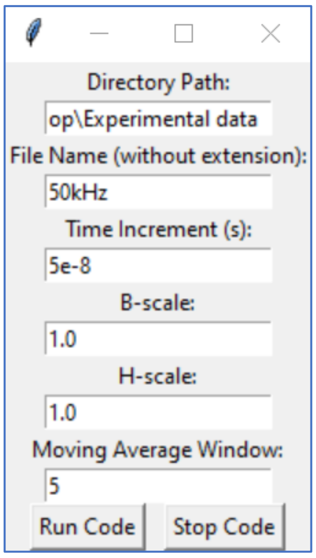
Your scale coefficients here (**1 by default**)

Used to smooth the BH curves (**5 by default**)

Change the file name and run the code. All output files will be rewritten.

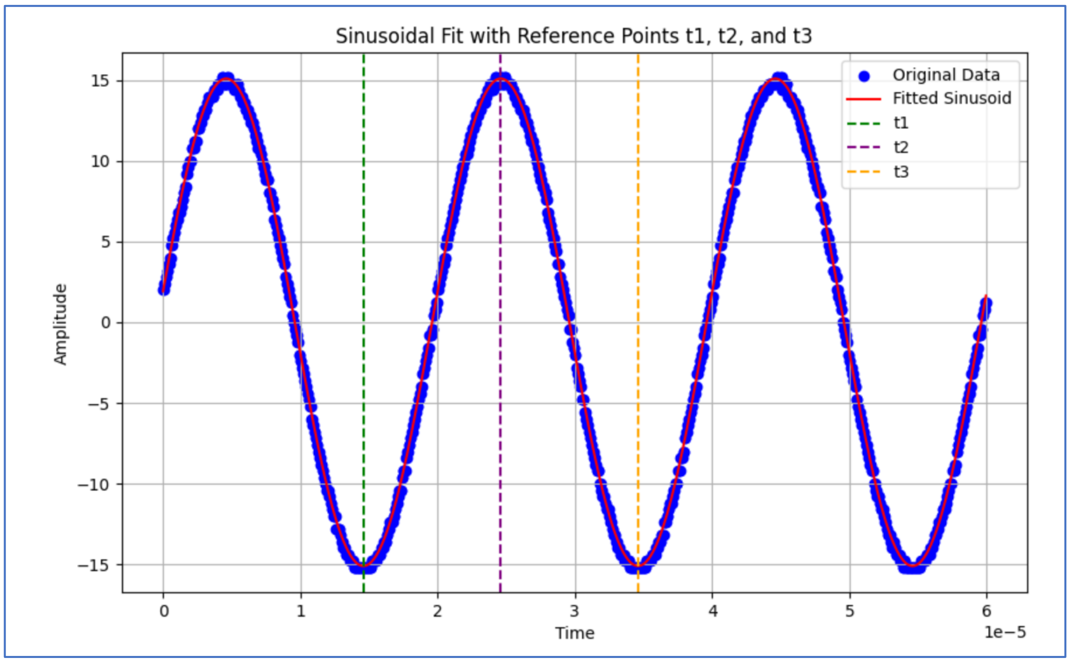
Stop the code

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

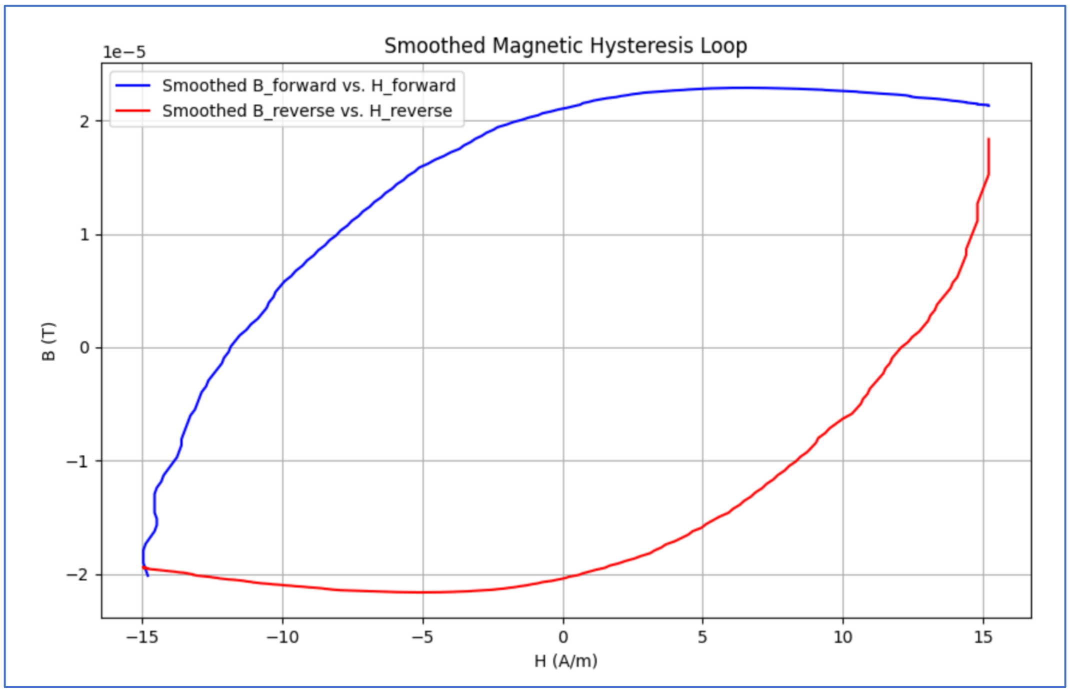


Directory Path:	op\Experimental data
File Name (without extension):	50kHz
Time Increment (s):	5e-8
B-scale:	1.0
H-scale:	1.0
Moving Average Window:	5

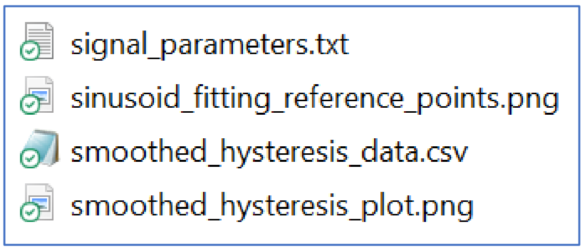
The initial figure generated by the code is displayed below. It will be saved in the PNG format. To proceed with the operation, you must close this figure.



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.



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



signal_parameters.txt
sinusoid_fitting_reference_points.png
smoothed_hysteresis_data.csv
smoothed_hysteresis_plot.png

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

Parameters saved in the **signal_parameters.txt** file:

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

Time increment = 5e-08 s
Fitted sinusoid amplitude = 15.099995113879167 (your units)
Fitted sinusoid frequency = 49992.53693715317 Hz
Fitted sinusoid phase = 0.12758812005225068 rads
Fitted sinusoid phase = 7.310260795002432 degrees

B-scale = 1.0
H-scale = 1.0

Reference time t1 = 1.4596053025498542e-05 s
Reference time t2 = 2.4597545860890395e-05 s
Reference time t3 = 3.459903869628225e-05 s

Reference index 1 = 291
Reference index 2 = 491
Reference index 3 = 691

```

BH.py (with GUI):

```python
#
# Digital BH-loop algorithm
# Project repository on GitHub: https://github.com/DYK-Team/Digital_BH-loop_algorithm
# 15/10/2023
#
# Authors:
# Ekaterina Nefedova, Dr. Mark Nemirovich, Dr. Nikolay Udanov, and Prof. Larissa Panina
# MISiS, Moscow, Russia, https://en.misis.ru/
#
# Dr. Dmitriy Makhnovskiy
# DYK+ team, United Kingdom, www.dykteam.com
#

import csv
import numpy as np
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt
import tkinter as tk

# Default input parameter values
pi = np.pi  # pi-constant 3.1415....
default_B_scale = 1.0  # This scale depends on the measurement units (V or mV) and the wire length and diameter
default_H_scale = 1.0  # This scale depends on the measurement units (V or mV) and the wire length and diameter
default_window_size = 5  # Moving Average Window. You can change from the GUI window.

# Function to run the code with the entered parameters
def run_code():
    directory_path = directory_path_entry.get()  # Copy and paste the directory path to the GUI window
    name = name_entry.get()  # Enter the file name without the CSV extension to the GUI window (e.g. 50kHz)
    time_increment = float(time_increment_entry.get())  # Enter the time increment (s) to the GUI window
    B_scale = float(B_scale_entry.get())
    H_scale = float(H_scale_entry.get())
    window_size = int(window_size_entry.get())

    # Full file name, including the directory path and the csv extension
    file_name = directory_path + '\\' +name + '.csv'

    # Data from the CSV file (two columns without header)
    data = np.genfromtxt(file_name, delimiter=',')
    response_values = data[:, 0]  # First column: response values which are proportional to the induction B
    sin_values = data[:, 1]  # Second column: sinusoid values which are proportional to the scanning magnetic field H
    N = len(sin_values)  # Number of points in each column

    # Time values based on the time increment
    time = np.arange(0, N * time_increment, time_increment)

    # Estimation for the sinusoid amplitude
    A0 = np.max(sin_values)

    # Scenarios for selecting sine wave vertices
    scenario = 1 if sin_values[0] >= 0 else 2

    # Skipping initial values until the sinusoid changes sign. After this, the search for vertices begins.
    i = 0
    while i <= N - 1 and ((scenario == 1 and sin_values[i] >= 0) or (scenario == 2 and sin_values[i] <= 0)):
        i += 1
    start = i  # Start index

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

    # Set of indices near the positive vertex of the sinusoid
    def pset(start, y_values):
        positive_set = []
        i = start
        while i <= N - 1 and y_values[i] >= 0:
            if A0 * 0.9 <= y_values[i] <= A0:
                positive_set.append(i)
            i += 1
        stop = i
        return np.array(positive_set), stop

    # Set of indices near the negative vertex of the sinusoid
    def nset(start, y_values):
        negative_set = []
        i = start
        while i <= N - 1 and y_values[i] <= 0:
            if -A0 <= y_values[i] <= -A0 * 0.9:
                negative_set.append(i)
            i += 1
        stop = i
        return np.array(negative_set), stop

    if scenario == 1:
        negative_set, stop = nset(start, sin_values)
        positive_set = pset(stop, sin_values)[0]
    elif scenario == 2:
        positive_set, stop = pset(start, sin_values)
        negative_set = nset(stop, sin_values)[0]

    # Average indices for the found sets of the negative and positive sinusoid vertices
    positive_set_aver = float(sum(positive_set) / len(positive_set))
    negative_set_aver = float(sum(negative_set) / len(negative_set))

    # Integer indices for the found negative and positive sinusoid vertices
    pvertex = int(positive_set_aver)
    nvertex = int(negative_set_aver)
```

```python

    # Estimations of the period (T0) and frequency (f0) of the sinusoid
    T0 = abs(positive_set_aver - negative_set_aver) * time_increment * 2
    f0 = 1 / T0

    # Improving A0: average amplitude of the sinusoid calculated from two vertices
    A0 = (sin_values[pvertex] - sin_values[nvertex]) / 2.0

    # Estimation of the sinusoid phase (ph0) in radians
    # Only positive phase will be used (anticlockwise rotation)
    qT = int(abs(pvertex - nvertex) / 2)  # Index difference for the quarter period
    value = max(-1.0, min(sin_values[0] / A0, 1.0))  # Normalising the zero-index value and clamping it to [-1, 1]
    phase = np.arcsin(value)  # Phase calculated from the arcsin function in the range [-pi/2, pi/2]
    if scenario == 1:  # First scenario sin_values[0] >= 0
        if sin_values[qT] >= 0:
            ph0 = phase  # First quarter
        else:
            ph0 = pi - phase  # Second quarter
    else:  # Second scenario sin_values[0] <= 0
        if sin_values[qT] <= 0:
            ph0 = pi - phase  # Third quarter
        else:
            ph0 = 2.0 * pi + phase  # Fourth quarter

    print('Estimated amplitude = ', A0)
    print('Estimated frequency = ', f0, ' Hz')
    print('Estimated phase = ', ph0, ' rads')
    print('Estimated phase = ', np.degrees(ph0), ' degrees')

    # Sinusoidal function used in the fitting
    def sinusoid(t, A, f, phase):
        return A * np.sin(2 * pi * f * t + phase)

    # Fitting the data (second column) to the sinusoid
    params, covariance = curve_fit(sinusoid, time, sin_values, p0=[A0, f0, ph0])  # [A0, f0, ph0] - initial values

    # Extracted fitting parameters
    A_fit, f_fit, ph_fit = params

    # Fitted sinusoid curve
    sinusoid_fit = sinusoid(time, A_fit, f_fit, ph_fit)
    ph_degrees = np.degrees(ph_fit)

    print('')
    print('Fitted amplitude = ', A_fit)
    print('Fitted frequency = ', f_fit, ' Hz')
    print('Fitted phase = ', ph_fit, ' rads')
    print('Fitted phase = ', ph_degrees, ' degrees')

    # Calculation of the reference time points t123 used in the numerical integration
    if scenario == 1:  # First scenario sin_values[0] >= 0
        t1 = (3.0 * pi / 2.0 - ph_fit) / (2.0 * pi * f_fit)
        t2 = (5.0 * pi / 2.0 - ph_fit) / (2.0 * pi * f_fit)
    else:  # Second scenario sin_values[0] <= 0
        t1 = (5.0 * pi / 2.0 - ph_fit) / (2.0 * pi * f_fit)
        t2 = (7.0 * pi / 2.0 - ph_fit) / (2.0 * pi * f_fit)
    t3 = t2 + 0.5 / f_fit

    print('')
    print('Reference time t1 = ', t1, ' s')
    print('Reference time t2 = ', t2, ' s')
    print('Reference time t3 = ', t3, ' s')

    # Indexes corresponding to the reference time moments
    refindex1 = int(t1 / time_increment)
    refindex2 = int(t2 / time_increment)
    refindex3 = int(t3 / time_increment)

    print('')
    print('Reference index 1 = ', refindex1)
    print('Reference index 2 = ', refindex2)
    print('Reference index 3 = ', refindex3)

    # Writing the parameters to the txt file
    with open(directory_path + '\\' + 'signal_parameters.txt', 'w') as file:
        file.write('\n')
        file.write('File name and directory {}\n'.format(file_name))
        file.write('\n')
        file.write('Time increment = {} s\n'.format(time_increment))
        file.write('Fitted sinusoid amplitude = {} (your units)\n'.format(A_fit))
        file.write('Fitted sinusoid frequency = {} Hz\n'.format(f_fit))
        file.write('Fitted sinusoid phase = {} rads\n'.format(ph_fit))
        file.write('Fitted sinusoid phase = {} degrees\n'.format(ph_degrees))
        file.write('\n')
        file.write('B-scale = {} \n'.format(B_scale))
        file.write('H-scale = {} \n'.format(H_scale))
        file.write('\n')
        file.write('Reference time t1 = {} s \n'.format(t1))
        file.write('Reference time t2 = {} s \n'.format(t2))
        file.write('Reference time t3 = {} s \n'.format(t3))
        file.write('\n')
        file.write('Reference index 1 = {} \n'.format(refindex1))
        file.write('Reference index 2 = {} \n'.format(refindex2))
        file.write('Reference index 3 = {} \n'.format(refindex3))

    # Plot the original data and the fitted curve
    plt.figure(figsize=(10, 6))
    plt.scatter(time, sin_values, label='Original Data', color='blue', marker='o')
    plt.plot(time, sinusoid_fit, label='Fitted Sinusoid', color='red')

    # Add vertical lines at t1, t2, and t3
    plt.axvline(x=t1, color='green', linestyle='--', label='t1')
    plt.axvline(x=t2, color='purple', linestyle='--', label='t2')
    plt.axvline(x=t3, color='orange', linestyle='--', label='t3')

    plt.xlabel('Time')
    plt.ylabel('Amplitude')
    plt.title('Sinusoidal Fit with Reference Points t1, t2, and t3')
    plt.legend()
    plt.grid(True)

    # Saving the plot as an image
    plt.savefig(directory_path + '\\' + 'sinusoid_fitting_reference_points.png')

    plt.show()

    # Forward integration of the voltage response between the reference indexes 1 and 2
    B_forward = []
    H_forward = []
    integral_value = 0.0
    for i in range(refindex1, refindex2):
        H_forward.append(sin_values[i])
        for i in range(refindex1, i):
            integral_value += 0.5 * (response_values[i] + response_values[i + 1]) * time_increment  # Trapezoid method
        B_forward.append(integral_value)

    # Reverse integration of the voltage response between the reference indexes 2 and 3
    B_reverse = []
    H_reverse = []
    integral_value = 0.0
    for i in range(refindex2, refindex3):
        H_reverse.append(sin_values[i])
        for i in range(refindex2, i):
            integral_value += 0.5 * (response_values[i] + response_values[i + 1]) * time_increment  # Trapezoid method
        B_reverse.append(integral_value)

    # Rescaling the magnetic induction B and the field H from the data values
    B_forward = np.array(B_forward) * B_scale
    H_forward = -np.array(H_forward) * H_scale
    lf = len(B_forward)  # Number of points in the forward BH curve
    B_reverse = np.array(B_reverse) * B_scale
    H_reverse = -np.array(H_reverse) * H_scale
    lr = len(B_reverse)  # Number of points in the reverse BH curve

    # Defining the concavity con_forward of the forward BH curve
    # B = af + bf * H is the straight line between the forward BH curve ends
    bf = (B_forward[lf - 1] - B_forward[0]) / (H_forward[lf - 1] - H_forward[0])
    af = (B_forward[0] * H_forward[lf - 1] - B_forward[lf - 1] * H_forward[0]) / (H_forward[lf - 1] - H_forward[0])
    # Direction of the concavity
    if (af + bf * (H_forward[lf - 1] - H_forward[0]) / 2) >= B_forward[int(lf / 2)]:
        con_forward = 'down'
    else:
        con_forward = 'up'

    # Defining the concavity con_reverse of the reverse BH curve
    # B = ar + br * H is the straight line between the reverse BH curve ends
    br = (B_reverse[lr - 1] - B_reverse[0]) / (H_reverse[lr - 1] - H_reverse[0])
    ar = (B_reverse[0] * H_reverse[lr - 1] - B_reverse[lr - 1] * H_reverse[0]) / (H_reverse[lr - 1] - H_reverse[0])
    # Direction of the concavity
    if (ar + br * (H_reverse[lr - 1] - H_reverse[0]) / 2) >= B_reverse[int(lr / 2)]:
        con_reverse = 'down'
    else:
        con_reverse = 'up'

     # Vertical shifts of the BH curves
    if con_forward == 'up':
        B_forward = B_forward + abs(B_forward[0] - B_forward[lf - 1]) / 2
    else:
        B_forward = B_forward - abs(B_forward[0] - B_forward[lf - 1]) / 2

    if con_reverse == 'up':
        B_reverse = B_reverse + abs(B_reverse[0] - B_reverse[lr - 1]) / 2
    else:
        B_reverse = B_reverse - abs(B_reverse[0] - B_reverse[lr - 1]) / 2

    # Function for computing the moving average
    def moving_average(data, window_size):
        cumsum = np.cumsum(data)
        cumsum[window_size:] = cumsum[window_size:] - cumsum[:-window_size]
        return cumsum[window_size - 1:] / window_size

    # Smoothing the data using moving averages
    B_forward_smoothed = moving_average(B_forward, window_size)
    H_forward_smoothed = moving_average(H_forward, window_size)
    B_reverse_smoothed = moving_average(B_reverse, window_size)
    H_reverse_smoothed = moving_average(H_reverse, window_size)

    # Creating the graph of smoothed curves
    plt.figure(figsize=(10, 6))
    plt.plot(H_forward_smoothed, B_forward_smoothed, label='Smoothed B_forward vs. H_forward', color='blue')
    plt.plot(H_reverse_smoothed, B_reverse_smoothed, label='Smoothed B_reverse vs. H_reverse', color='red')
    plt.xlabel('H (A/m)')
    plt.ylabel('B (T)')
    plt.title('Smoothed Magnetic Hysteresis Loop')
    plt.legend()
    plt.grid(True)

    # Saving the data and smoothed curves to a CSV file
    data = np.column_stack((H_forward_smoothed, B_forward_smoothed, H_reverse_smoothed, B_reverse_smoothed))
    header = ['H_forward (A/m)', 'B_forward_smoothed (T)', 'H_reverse (A/m)', 'B_reverse_smoothed (T)']

    with open(directory_path + '\\' + 'smoothed_hysteresis_data.csv', 'w', newline='') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow(header)
        writer.writerows(data)

    # Saving the plot as an image
    plt.savefig(directory_path + '\\' + 'smoothed_hysteresis_plot.png')

    plt.show()

# Create a function to stop the code execution
def stop_code():
    quit()

# Main GUI window
root = tk.Tk()
root.title("Input Parameters")

# Labels and entry fields for input parameters
directory_path_label = tk.Label(root, text="Directory Path:")
directory_path_label.pack()
directory_path_entry = tk.Entry(root)
directory_path_entry.pack()

name_label = tk.Label(root, text="File Name (without extension):")
name_label.pack()
name_entry = tk.Entry(root)
name_entry.pack()

time_increment_label = tk.Label(root, text="Time Increment (s):")
time_increment_label.pack()
time_increment_entry = tk.Entry(root)
time_increment_entry.pack()

B_scale_label = tk.Label(root, text="B-scale:")
B_scale_label.pack()
B_scale_entry = tk.Entry(root)
B_scale_entry.insert(0, default_B_scale)  # Default value
B_scale_entry.pack()

H_scale_label = tk.Label(root, text="H-scale:")
H_scale_label.pack()
H_scale_entry = tk.Entry(root)
H_scale_entry.insert(0, default_H_scale)  # Default value
H_scale_entry.pack()

window_size_label = tk.Label(root, text="Moving Average Window:")
window_size_label.pack()
window_size_entry = tk.Entry(root)
window_size_entry.insert(0, default_window_size)  # Default value
window_size_entry.pack()

# Frame to hold the buttons in one row
button_frame = tk.Frame(root)
button_frame.pack()

# "Run Code" button with some padding to the right
run_button = tk.Button(button_frame, text="Run Code", command=run_code)
run_button.pack(side=tk.LEFT, padx=5)  # Adjust the padx value as needed

# "Stop Code" button with some padding to the left
stop_button = tk.Button(button_frame, text="Stop Code", command=stop_code)
stop_button.pack(side=tk.LEFT, padx=5)  # Adjust the padx value as needed

# Main event loop
root.mainloop()

```