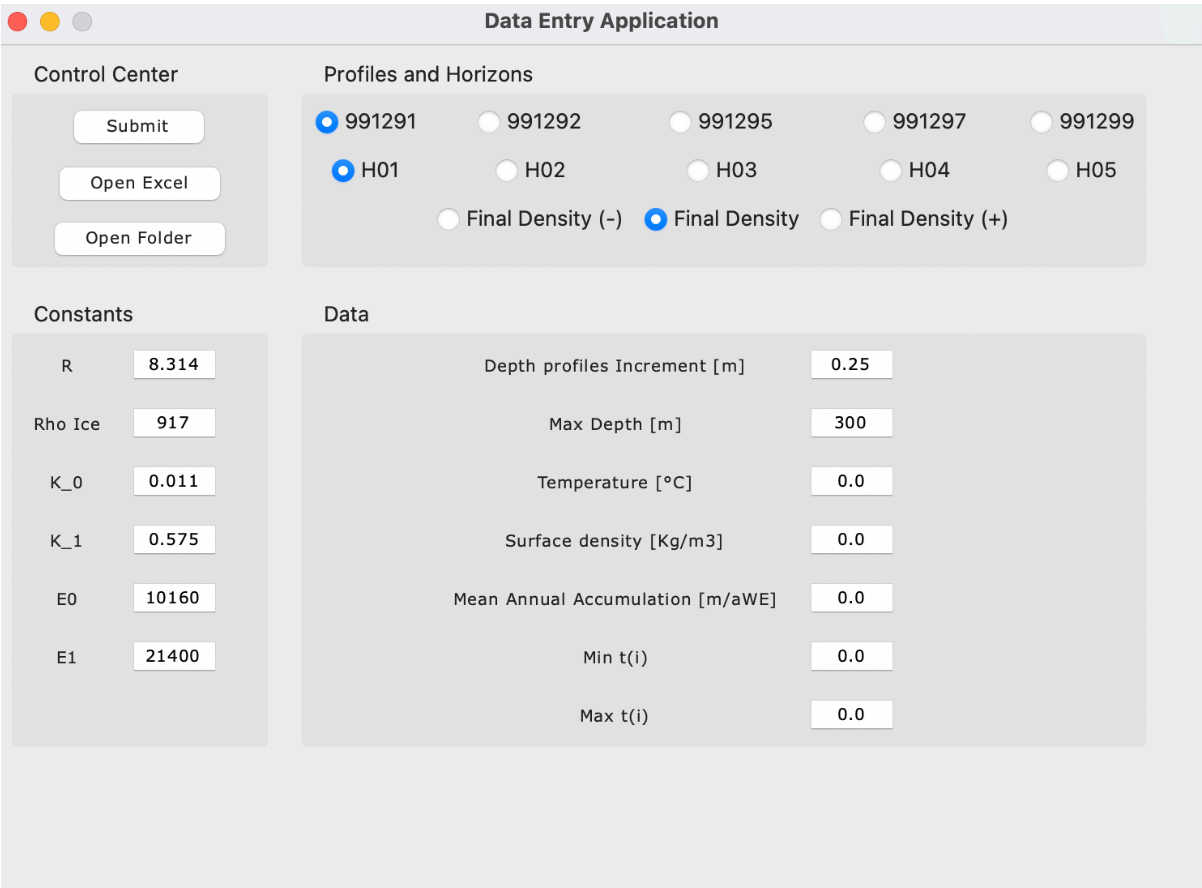
Python code for estimation of densities using Herron and Langway method.

The python code was written in order to estimate the densities using Herron and Langway method of density estimations. Since, the calculations are complicated and interlinked, it was easier to use python for the automatization. The given code here also calculates the errors in order to assist in the error calculations and estimations later.
First, the GUI is explained and then the requirements for the input file ("Template File"), so that the code finds the respective column and writes the information correctly. It is important for the person using this code to ensure that this "Template File" is accordingly explained below. If the user wants they can also make changes for the headers in the input file, but the code to create the Template file by running the script is also attached below.



The GUI for the data management is made where the user can select what line and horizon they need to use to do the calculations. There are columns to insert values for:

1. Depth profile:

The user can define how long the depth profile is needed and at what increment. The default value is 0.25 meters for the increment and 300 meters for the total length of the profile.

2. Temperature:

The temperature can be added in degree Celsius and the code will automatically use the value in Kelvins since the Herron Langway requires the temperature to be in Kelvins.

3. Surface density:

The surface density is also set default as 0.0 because it can change everywhere along the profile, so, the user can be flexible in using different values along the traverse for each specific profile. The surface density input should be in kg/m^3.

4. Mean Annual Accumulation:

Since, the accumulation is also changing for every part of the traverse it is also used in correspondence to the Surface density of that specific part of the traverse. The unit should be in meters per annum Water Equivalent.

5. Min t(i) and Max t(i)

This value of the minimum and maximum Two way travel time from the picked horizons is also changing along the profile for each horizon in each profile, so it should be corelated with the picked data and inserted accordingly for the calculation of the mean Velocity with respect to the depth (Vm(z)).

2. Constants:

The constants tab on the right of the GUI already have default values which are used for the calculations of the density.

3. Final density row:

The final density row below the horizons row can be selected to calculate the density for each type of error limit, for example, if the user wants to calculate the density with using the surface density with the lower end of the error, it can be calculated by inserting the value of surface density down below and the final density(-) option should be selected so that the output file displays the columns correctly.

4. Control center:

This section in the top right is self-explanatory. Submit button should be used first in order to calculate the densities. Later the other two buttons can be used for what they are meant for.

Requirements for the template file:

The user has two options- either they can run the code directly posted here or they can create their own template file with the following headers starting from column A and moving ahead for each column in row 1:

depth(m),
final density (-),
final density,
final density (+),

time(ms){horizon_name},
dist. from Neumayer(km)_{profile_number},
C (-),
C,
C (+),
t(i) (-),
t(i),
t(i) (+),
Et(i)(cumulative) (-),
Et(i)(cumulative),
Et(i)(cumulative) (+),
Vm(z) (-),
Vm(z),
Vm(z) (+),
mean V (-),
mean V,
mean V (+),
depth(m){horizon_name} (-),
depth(m){horizon_name},
depth(m){horizon_name} (+),
mass(kg) (-),
mass(kg),
mass(kg) (+),
cumulative mass (-),
cumulative mass,
cumulative mass (+),
SWE(mWE) (-),
SWE(mWE),
SWE(mWE) (+),
{horizon_name}(SWE) (-),
{horizon_name}(SWE),
{horizon_name}(SWE) (+),
Age (-),
Age',
Age (+),
rho_Z_below_550,
rho_Z_above_550,
t_550,
t_rho.

```
import pandas as pd


class Excel_Creator:
    def __init__(self, file_path):
```

```
        self.profiles_list = [991291, 991292, 991295, 991297,
991299]
        self.Horizons_list = ["H01", "H02", "H03", "H04", "H05"]
        self.file_path = file_path
        self.create_excel_with_empty_sheets()

    def create_excel_with_empty_sheets(self):
        """
        Creates an Excel file with sheets named by combining
elements from Horizons_list with the last two digits of
        each corresponding number in profiles_list.

        Args:
        file_path (str): The path to the Excel file to be created.
        profiles_list (list): List of profile numbers.
        Horizons_list (list): List of Horizon identifiers.
        """
        # Generate sheet names by combining Horizon codes with the
last two digits of profile codes
        sheet_names = [f"{horizon}_{str(profile)[-2:]}"
                       for profile in self.profiles_list for horizon
in self.Horizons_list]

        # Create an Excel file with the specified empty sheets
        with pd.ExcelWriter(self.file_path, engine='xlsxwriter') as
writer:
            workbook = writer.book  # Get the XlsxWriter Workbook
object

            # Define the cell format for headers
            header_format = workbook.add_format({'bold': True,
'align': 'center', 'valign': 'vcenter'})

            for sheet_name in sheet_names:
                # Parse Horizon and profile from the sheet name
                horizon, profile_suffix = sheet_name.split('_')
                headers = self.generate_headers(horizon,
profile_suffix)
                # Create an empty DataFrame with these headers
                df = pd.DataFrame(columns=headers)

                # Create an empty DataFrame for each sheet
                df.to_excel(writer, sheet_name=sheet_name,
index=False)

                # Access the XlsxWriter worksheet object and format
header cells
                worksheet = writer.sheets[sheet_name]
```

```
                for col_num, header in enumerate(headers):
                    max_len = max(len(header), 10)  # Adjust column
width if necessary
                    worksheet.set_column(col_num, col_num, max_len)
# Set column width
                    worksheet.write(0, col_num, header,
header_format)  # Write header with format

    @staticmethod
    def generate_headers(horizon, profile):
        """
                Generates a list of headers based on the Horizon and
profile suffix.

                Args:
                horizon (str): The Horizon part of the sheet name.
                profile_suffix (str): The profile suffix part of the
sheet name.

                Returns:
                list: List of formatted header strings.
                """
        return [
            'depth(m)', f'final density (-)', 'final density',
'final density (+)',
            f'time(ms){horizon}', f'dist. from
Neumayer(km)_{profile}',
            'C (-)', 'C', 'C (+)', 't(i) (-)', 't(i)', 't(i) (+)',
            'Et(i)(cumulative) (-)', 'Et(i)(cumulative)',
'Et(i)(cumulative) (+)',
            'Vm(z) (-)', 'Vm(z)', 'Vm(z) (+)', 'mean V (-)', 'mean
V', 'mean V (+)',
            f'depth(m){horizon} (-)', f'depth(m){horizon}',
f'depth(m){horizon} (+)',
            'mass(kg) (-)', 'mass(kg)', 'mass(kg) (+)',
            'cumulative mass (-)', 'cumulative mass', 'cumulative
mass (+)',
            'SWE(mWE) (-)', 'SWE(mWE)', 'SWE(mWE) (+)',
            f'{horizon}(SWE) (-)', f'{horizon}(SWE)',
f'{horizon}(SWE) (+)',
            f'Age (-)', f'Age', f'Age (+)',
            'rho_Z_below_550', 'rho_Z_above_550', 't_550', 't_rho'
        ]


Excel_Creator('Template_file.xlsx')
```

The backend GUI script utilizes the ‘tkinter’ library for building a graphical user interface (GUI) application, along with other libraries such as ‘pandas’ for data manipulation, ‘openpyxl’ for working with Excel files, and ‘subprocess’ for running external commands, aimed at performing various tasks such as data processing, file manipulation, and running subprocesses concurrently using threading. ‘shutil’ is used for high-level file operations like copying, moving and deleting files and directories.

```
import tkinter as tk
from tkinter import ttk, messagebox
import pandas as pd
import openpyxl
import shutil
import numpy as np
import threading
import os
import subprocess
import sys


class DataEntryApp:
    def __init__(self, root):
        self.Excel_button = None
        self.submit_button = None
        self.output_excel_data = None
        self.final_density_header = None
        self.sheet_name = None
        self.Density_type = None
        self.Horizon = None
        self.profile = None
        self.profiles_list = [991291, 991292, 991295, 991297,
991299]
        self.Horizons_list = ["H01", "H02", "H03", "H04", "H05"]
        self.density_types_list = [" (-)", "", " (+)"]
        self.output_excel_path = "calculation_results.xlsx"
        self.template_file_path = "Template_file.xlsx"
        self.time_dist_excel_path = "time_and_distance.xlsx"
        self.root = root
        self.root.title("Data Entry Application")
        self.root.geometry('750x525')  # Width x Height
        self.root.resizable(False, False)

        # DoubleVar variables for each parameter
        self.R_var = tk.DoubleVar(value=8.314)
        self.rho_ice_var = tk.DoubleVar(value=917)
```

```
        self.K_0_var = tk.DoubleVar(value=0.011)
        self.K_1_var = tk.DoubleVar(value=0.575)
        self.E0_var = tk.DoubleVar(value=10160)
        self.E1_var = tk.DoubleVar(value=21400)
        self.depth_increment_var = tk.DoubleVar(value=0.25)
        self.depth_max_var = tk.DoubleVar(value=300)
        self.Temperature_Celsius_var = tk.DoubleVar()
        self.rho_0_var = tk.DoubleVar()
        self.accumulation_var = tk.DoubleVar()
        self.min_T_i_var = tk.DoubleVar()
        self.max_T_i_var = tk.DoubleVar()

        self.R = None
        self.rho_ice = None
        self.K_0 = None
        self.K_1 = None
        self.E0 = None
        self.E1 = None
        self.depth_increment = None
        self.depth_max = None
        self.Temperature_Celsius = None
        self.rho_0 = None
        self.accumulation = None
        self.min_T_i = None
        self.max_T_i = None
        self.Temperature_Kelvin = None

        self.Profile_option = tk.IntVar(value=1)
        # self.Profile_option.set(1)  # Set default to the first
radio button
        self.Horizon_option = tk.IntVar()
        self.Horizon_option.set(1)  # Set default to the first radio
button
        self.Density_type_option = tk.IntVar()
        self.Density_type_option.set(2)  # Set default to the first
radio button
        # Configure custom styles
        style = ttk.Style()
        style.configure("Custom.TButton", font=('Verdana', 10))
        style.configure("Custom.TLabel", font=('Verdana', 10),
padding=5)

        self.setup_ui()
```

- To initialise to set up the basic structure and parameters needed for the data entry application’s GUI, a class is defined named ‘DataEntryApp’.

- The ‘__init__’ method initialise the various attributes and sets up the GUI window using “tkinter”

- Several attributes are setup such as buttons, variables to hold data, lists for profiles, horizons and densities, and paths to the Excel files.
- The Tkinter window is configures with a title, dimension and the resizing is disabled.
- DoubleVar variables are used for data entry parameters. They hold the default values.
- Custom styles for buttons and labels are configured using ttk.Style().
- The ‘setup_ui’ method is called to set up the GUI components.

```
def setup_ui(self):
    # Create a label frame for organized grouping of labels and
entries
    start_frame = ttk.LabelFrame(self.root, text="Profiles and
Horizons")
    start_frame.grid(row=0, column=1, padx=10, pady=10,
sticky='nsew')
    for i in range(5):
        profile = self.profiles_list[i]
        radio = ttk.Radiobutton(start_frame,
            variable=self.Profile_option, value=i + 1,  # Value
starts at 1 and goes up
            takefocus=0, style='CustomTRadiobutton.TRadiobutton',
            text=profile)  # Profile numbers increment by 1
        # Arrange the radiobuttons in a row; change 'row' to
'column=i' to place them horizontally
        radio.grid(row=0, column=i, pady=5, padx=5)

    for i in range(5):
        radio = ttk.Radiobutton(start_frame,
            variable=self.Horizon_option, value=i + 1,
            takefocus=0, style='CustomTRadiobutton.TRadiobutton',
            text=self.Horizons_list[i])
        # Arrange the radiobuttons in a row; change 'row' to
'column=i' to place them horizontally
        radio.grid(row=1, column=i, pady=5, padx=5)

    for i in range(len(self.density_types_list)):
        tag = self.density_types_list[i]
        radio = ttk.Radiobutton(start_frame,
                                variable=self.Density_type_option,
value=i + 1,
                                takefocus=0,
```

```
style='CustomTRadiobutton.TRadiobutton',
                                text=f"Final Density{tag}")
        # Arrange the radiobuttons in a row; change 'row' to
'column=i' to place them horizontally
        radio.grid(row=2, column=i + 1, pady=5, padx=5)

    constants_frame = ttk.LabelFrame(self.root, text="Constants")
    constants_frame.grid(row=1, column=0, padx=10, pady=10,
sticky='nsew')

    labels_text = ["R", "Rho Ice", "K_0", "K_1", "E0", "E1"]
    variables = [self.R_var, self.rho_ice_var, self.K_0_var,
self.K_1_var, self.E0_var, self.E1_var]

    # Creating labels and entries inside the frame
    for i, (text, var) in enumerate(zip(labels_text, variables)):
        # Label for each entry
        label = ttk.Label(constants_frame, text=text,
style="Custom.TLabel")
        label.grid(row=i, column=0, pady=5, padx=5)

        # Entry widget for each parameter
        entry = ttk.Entry(constants_frame, font=('Verdana', 10),
width=6, textvariable=var, justify='center')
        entry.grid(row=i, column=1, pady=5, padx=5)

    changable_frame = ttk.LabelFrame(self.root, text="Data")
    changable_frame.grid(row=1, column=1, padx=10, pady=10,
sticky='nsew')

    labels_text = ["Depth profiles Increment [m]", "Max Depth [m]",
"Temperature [°C]",
                   "Surface density [Kg/m3]", "Mean Annual
Accumulation [m/aWE]",
                   "Min t(i)", "Max t(i)"]
    variables = [self.depth_increment_var, self.depth_max_var,
self.Temperature_Celsius_var,
                  self.rho_0_var, self.accumulation_var,
self.min_T_i_var, self.max_T_i_var]
    # Creating labels and entries inside the frame
    for i, (text, var) in enumerate(zip(labels_text, variables)):
        # Label for each entry
        label = ttk.Label(changable_frame, text="\t",
style="Custom.TLabel")
        label.grid(row=i, column=0, pady=5, padx=5)

        label = ttk.Label(changable_frame, text=text,
style="Custom.TLabel")
```

```
        label.grid(row=i, column=1, pady=5, padx=5)

        # Entry widget for each parameter
        entry = ttk.Entry(changable_frame, font=('Verdana', 10),
width=6, textvariable=var, justify='center')
        entry.grid(row=i, column=2, pady=5, padx=5)

    control_frame = ttk.LabelFrame(self.root, text="Control Center")
    control_frame.grid(row=0, column=0, padx=10, pady=10,
sticky='nsew')

    label = ttk.Label(control_frame, text="", style="Custom.TLabel")
    label.grid(row=0, column=0, pady=5, padx=5)

    # Submit button at the bottom spanning across all columns
    self.submit_button = ttk.Button(control_frame, text="Submit",
                                   style="Custom.TButton",
command=self.submit_data)
    self.submit_button.grid(row=0, column=1, pady=5)

    self.Excel_button = ttk.Button(control_frame, text="Open Excel",
                                   style="Custom.TButton",
command=self.open_excel)
    self.Excel_button.grid(row=1, column=1, pady=5)

    Folder_button = ttk.Button(control_frame, text="Open Folder",
                              style="Custom.TButton",
command=self.open_folder)
    Folder_button.grid(row=2, column=1, pady=5)

    label = ttk.Label(control_frame, text="", style="Custom.TLabel")
    label.grid(row=0, column=2, pady=5, padx=5)
```

The setup_ui basically sets up the entire GUI interface.

- The label frames are created to organize and group labels, entries and buttons into different sections within the GUI window.

- Radio buttons are created for selecting profiles and horizons. Same for density selections.

- Inside the “constants” label frame, entry widgets are created for entering constant parameters and these entries are linked to the DoubleVar variables inititalised earlier.

- Inside the “Data” label frame, changeable data such as depth, surface density, etc., can be entered which are again linked to respective DoubleVar variables.
- Control Center label frame is created to house the control buttons.
- Each button is associated with a command to execute specific actions when clicked. (Example ‘submit_data()’ method).

```
def submit_data(self):

    self.R = float(self.R_var.get())
    self.rho_ice = float(self.rho_ice_var.get())
    self.K_0 = float(self.K_0_var.get())
    self.K_1 = float(self.K_1_var.get())
    self.E0 = float(self.E0_var.get())
    self.E1 = float(self.E1_var.get())
    self.depth_increment = float(self.depth_increment_var.get())
    self.depth_max = float(self.depth_max_var.get())
    self.Temperature_Celsius =
float(self.Temperature_Celsius_var.get())
    self.rho_0 = float(self.rho_0_var.get())
    self.accumulation = float(self.accumulation_var.get())
    self.min_T_i = float(self.min_T_i_var.get())
    self.max_T_i = float(self.max_T_i_var.get())

    profile_option = self.Profile_option.get()
    profile = str(self.profiles_list[profile_option - 1])
    self.profile = profile[-2:]

    Horizon_option = self.Horizon_option.get()
    self.Horizon = f"H0{Horizon_option}"
    Density_type = self.Density_type_option.get()
    self.Density_type = self.density_types_list[Density_type - 1]

    self.sheet_name = f'{self.Horizon}_{self.profile}'
    print(self.sheet_name)
    self.final_density_header = f'final density{self.Density_type}'
    self.first_calc()

    # get time and distance from Excel file:
    self.get_excel_input_data()

    # Start SWE calculations:
    self.second_calc()
    print("Done")

def open_excel(self):
```

```
    os.startfile(self.output_excel_path)

def open_folder(self):
    folder_path =
os.path.dirname(os.path.realpath(self.output_excel_path))
    if sys.platform == "win32":
        os.startfile(folder_path)
    elif sys.platform == "darwin":  # macOS
        subprocess.run(["open", folder_path])
    elif sys.platform.startswith("linux"):
        subprocess.run(["xdg-open", folder_path])

def get_excel_input_data(self):
    sheet_name = f"9912{self.profile}"
    data_input = pd.read_excel(self.time_dist_excel_path,
sheet_name=sheet_name)
    self.Update_excel_file(f"dist. from
Neumayer(km)_{self.profile}",
                           data_input[f"dist. from
Neumayer(km)_{self.profile}"])
    self.Update_excel_file(f"time(ms){self.Horizon}",
data_input[f"time(ms){self.Horizon}"])
```

- The data is retrieved from the inputs by the user from the GUI and are stored in the corresponding variables. (converted to floats before storing).

- The profiles, Horizons, density type are also stored in the respective “self.xx” variables.

- The sheet name are constructed based on the selected radio button for the selected horizon and the profile number.

- The function calls like ‘first_calc()’, ‘get_excel_input_data()’ and ‘second_calc()’ are called to perform calculations and retrieving of data.

- Wait for the program to display “Done.”

- The ‘open_excel()’ open the file using ‘os.startfile()’ independent of the platform and uses the default application to open “.xlsx’ files.

- The folder can be opened using ‘os.startfile()’ for Windows and the subprocess commands for macOS and Linux.

- The ‘get_excel_input_data()’ method reads specific data from the Excel file using ‘pd.read_excel()’ function from the pandas library. The data is read according to the selections in the GUI.

- The 'Update_excel_file()' method is utilised to update the specific colums in the output Excel file with the data retrieved from the input Excel file.

```
def first_calc(self):
    print("Calculating..")
    self.Temperature_Kelvin = self.Temperature_Celsius + 273
    depth_profile = [x * self.depth_increment for x in
range(int(self.depth_max / self.depth_increment) + 1)]
    if os.path.exists(self.output_excel_path):
        self.Update_excel_file('depth(m)', depth_profile)

    else:
        shutil.copyfile(self.template_file_path,
self.output_excel_path)
        self.Update_excel_file('depth(m)', depth_profile)

    K_0_calculated = self.K_0 * np.exp(-self.E0 / (self.R *
self.Temperature_Kelvin))
    Z_0 = np.exp(self.rho_ice * K_0_calculated *
self.output_excel_data['depth(m)'] +
                 np.log(self.rho_0 / (self.rho_ice - self.rho_0)))
    rho_Z_below_550 = (self.rho_ice * Z_0) / (1 + Z_0)
    K_1_calculated = self.K_1 * np.exp(-self.E1 / (self.R *
self.Temperature_Kelvin))

    H_550 = (1 / (self.rho_ice * K_0_calculated)) * (
                np.log(550 / (self.rho_ice - 550)) -
np.log(self.rho_0 / (self.rho_ice - self.rho_0)))

    Z_1 = np.exp((self.rho_ice * K_1_calculated *
(self.output_excel_data['depth(m)'] - H_550))
                 / np.sqrt(self.accumulation)) + np.log(550 /
(self.rho_ice - 550))

    rho_Z_above_550 = (self.rho_ice * Z_1) / (1 + Z_1)
    t_550 = ((1 / (K_0_calculated * self.accumulation * 1000)) *
             np.log((self.rho_ice - self.rho_0) / (self.rho_ice -
rho_Z_below_550)))
    age_550 = ((1 / (K_0_calculated * self.accumulation * 1000)) *
             np.log((self.rho_ice - self.rho_0) / (self.rho_ice -
550)))
    # t_rho = ((np.log((self.rho_ice - 550) / (self.rho_ice -
rho_Z_above_550)) /
    #           K_1_calculated * np.sqrt(self.accumulation)) + t_550)
/ 1000
    # t_x = K_1_calculated * np.sqrt(self.accumulation)
    # t_1 = self.rho_ice - 550
    # t_2 = self.rho_ice - rho_Z_above_550
```

```
    # t_rho = np.log(t_1 / t_2) / t_x
    t_rho = (np.log((self.rho_ice - 550) / (self.rho_ice -
rho_Z_above_550)) /
             (K_1_calculated * np.sqrt(self.accumulation))) + t_550
    t_rho /= 1000
    self.Update_excel_file('t_550', t_550)
    self.Update_excel_file('t_rho', t_rho)
    self.Update_excel_file('rho_Z_below_550', rho_Z_below_550)
    self.Update_excel_file('rho_Z_above_550', rho_Z_above_550)

    # Filter values under 550 from 'rho_Z_below_550'
    below_550 =
self.output_excel_data['rho_Z_below_550'][self.output_excel_data['rh
o_Z_below_550'] < 550]

    # Filter values 550 or above from 'rho_Z_above_550'
    above_or_equal_550 =
self.output_excel_data['rho_Z_above_550'][self.output_excel_data['rh
o_Z_above_550'] >= 550]

    # Concatenate these two series into one, ignoring the original
index to avoid duplicate indexes
    final_density = pd.concat([below_550, above_or_equal_550],
ignore_index=True)
    self.Update_excel_file(self.final_density_header, final_density)
    self.output_excel_data.drop(columns=['rho_Z_below_550',
'rho_Z_above_550'])

    # Filter values under 550 from 'rho_Z_below_550'
    below_550 =
self.output_excel_data['t_550'][self.output_excel_data['t_550'] <
age_550]

    # Filter values 550 or above from 'rho_Z_above_550'
    above_or_equal_550 =
self.output_excel_data['t_rho'][self.output_excel_data['t_rho'] >=
age_550]

    # Concatenate these two series into one, ignoring the original
index to avoid duplicate indexes
    age = pd.concat([below_550, above_or_equal_550],
ignore_index=True)

    # Add the new 'final density' series to the DataFrame
    self.Update_excel_file(f"Age{self.Density_type}", age)

def second_calc(self):
    C = (3 * (10 ** 8)) / (10 ** 9 * (1 + 0.000845 *
```

```
self.output_excel_data[f'final density{self.Density_type}']))
    t_i = 0.25 / C

    self.Update_excel_file(f'C{self.Density_type}', C)
    self.Update_excel_file(f't(i){self.Density_type}', t_i)
    Et_i = t_i.cumsum()
    self.Update_excel_file(f'Et(i)(cumulative){self.Density_type}',
Et_i)
    Vm_z = ((0.25 + self.output_excel_data[f'depth(m)']) /

self.output_excel_data[f'Et(i)(cumulative){self.Density_type}'])
    self.Update_excel_file(f'Vm(z){self.Density_type}', Vm_z)

    # Find the closest values for input values in the 'Vm(z)' column
    closest_value_1 =
self.find_closest_and_fetch(self.output_excel_data,

f"Et(i)(cumulative){self.Density_type}",

f'Vm(z){self.Density_type}', self.min_T_i)
    closest_value_2 =
self.find_closest_and_fetch(self.output_excel_data,

f"Et(i)(cumulative){self.Density_type}",

f'Vm(z){self.Density_type}', self.max_T_i)
    mean_V = (closest_value_1 + closest_value_2) / 2
    self.output_excel_data[f'mean V{self.Density_type}'] = mean_V

    depth = mean_V *
(self.output_excel_data[f'time(ms){self.Horizon}'] / 2)

self.Update_excel_file(f'depth(m){self.Horizon}{self.Density_type}',
depth)
    mass = 0.25 * self.output_excel_data[f'final
density{self.Density_type}']
    self.Update_excel_file(f'mass(kg){self.Density_type}', mass)

    cumulative_mass =
self.output_excel_data[f'mass(kg){self.Density_type}'].cumsum()
    self.Update_excel_file(f'cumulative mass{self.Density_type}',
cumulative_mass)
    SWE = self.output_excel_data[f'cumulative
mass{self.Density_type}'] / 1000
    self.Update_excel_file(f'SWE(mWE){self.Density_type}', SWE)

    H_SWE =
(self.output_excel_data[f'depth(m){self.Horizon}{self.Density_type}'
```

```
].
        apply(lambda x:
self.output_excel_data.loc[self.match_lookup(x,
self.output_excel_data['depth(m)']),
    f'SWE(mWE){self.Density_type}']
        if self.match_lookup(x, self.output_excel_data['depth(m)'])
is not None else None))

self.Update_excel_file(f'{self.Horizon}(SWE){self.Density_type}',
H_SWE)
```

- In the 'first_calc()' method, the calculation are carried out in a step wise fashion, starting with conversion of Temperaure into Kelvin, then the depth profile creation, followed by the calculations for different variables. Filters data based on the specific conditions and concatenates them into a final density and age series followed by updating the excel sheet.

- The 'second_calc()' calculated the other values which are derived using the variables calculated in the 'first_calc()' and then updates the excel sheet again.

```
@staticmethod
def find_closest_and_fetch(data, search_column, fetch_column,
reference_value):
    """
    Finds the closest value to the specified value in the given
DataFrame column and
    fetches a corresponding value from another column in the same
row.

    Args:
    data (pd.DataFrame): The pandas DataFrame containing the data.
    Search_column (str): The column name to search for the closest
value.
    Fetch_column (str): The column name from which to fetch the
value.
    Reference_value (float): The value to find the closest to.

    Returns:
    float: The value from the fetch_column of the row with the
closest value in the search_column.
    """
    # Calculate the absolute differences from the reference value in
the search column
```

```
    abs_diff = (data[search_column] - reference_value).abs()

    # Find the index of the row with the minimum difference
    closest_index = abs_diff.idxmin()

    # Fetch and return the value from the fetch column at the found
index
    return data.at[closest_index, fetch_column]

@staticmethod
def match_lookup(value, lookup_series):
    """ Find the largest value less than or equal to the lookup
value. """
    return lookup_series[lookup_series <= value].last_valid_index()

def Update_excel_file(self, header, content_list):
    """
    Updates or appends data under a specific header in a specified
sheet.

    Args:
    header (str): The header name under which the data will be
updated.
    content_list (list): The list of contents to place under the
header.
    """
    # Load the workbook and the specific sheet
    wb = openpyxl.load_workbook(self.output_excel_path)
    ws = wb[self.sheet_name]

    # Find the column for the specified header
    column_letter = None
    for cell in ws[1]:  # The first row contains headers
        if cell.value == header:
            column_letter = cell.column_letter
            break

    if not column_letter:
        print(f"Header '{header}' not found in
'{self.sheet_name}'.")
        return

    # Update or append data under the found header column
    for i, value in enumerate(content_list, start=2):  # Start at
row 2 to skip header
        ws[f"{column_letter}{i}"].value = value

    # Save the workbook
```

```
    wb.save(self.output_excel_path)
    self.output_excel_data = pd.read_excel(self.output_excel_path,
sheet_name=self.sheet_name)
```

- The static methods are utility functions used for data manipulation and Excel file handling within the 'DataEntryApp' class.

- The 'find_closest_and_fetch()' method finds the closest value (minimum absolute difference) to the reference value and fetches the corresponding value from another column.

- The 'match_lookup()' method finds the largest values less than or equal to the lookup values in the given series and after these two eventually the excel sheet is updated again.

The GUI is run by the entry point script called 'main.py':

```
import tkinter as tk
import GUI

root = tk.Tk()
GUI.DataEntryApp(root)
root.mainloop()
```

- The libraries are imported like 'tkinter' and 'GUI'. The 'GUI' module contains the 'DataEntryApp' class from the 'GUI.py' script explained above. The 'tkinter' library is used to create the GUI.

- 'root = tk.Tk()' creates a root window for the GUI application containing the widgets and all the elements of the GUI.

- 'GUI.DataEntryApp(root)' creates an instance of the 'DataEntryApp' class from the 'GUI' module, passing the root window as an argument. This initializes and sets up the GUI application with all its components and functionality.

- 'root.mainloop()' enters the main event loop of the Tkinter application. This loop hears for events like keyboard inputs, window resizing, button clicks, etc. and dispatches them to the appropriate event handlers. The application remains in the loop until the user closes the window, at which point the loop exits and the script terminates.

In case the user needs to use this code, it is possible to contact me via email:
aman.gupta1602@gmail.com

Thanks and have fun!