```python
##################################################################################
# Created by Ben Marosites
# marosite@email.sc.edu
# For GEOL 365/599
# Fall 2025 - December 01
# These are functions I created or modified to use in my analysis.
# Most is related to extracting and processing data.
##################################################################################

import pandas as pd
import numpy as np
import math
import datetime as dt
from matplotlib import pyplot as plt

import os
import shutil
import glob

from netCDF4 import Dataset
import xarray as xr

from scipy.interpolate import griddata
from scipy.signal import argrelextrema, find_peaks

def extract_tar_files(directory):
    '''
    This was just cut from one of my notebooks. Should only be run once to extract data
    and move it to the correct final location. This function will need to be redone, but
    for now, it will be incomplete.

    Parameters
    ----------

    Returns
    ----------

    Examples
    ----------

    '''
    extract = False # ONLY NEED TO RUN THIS ONCE, SET TO FALSE AFTER DATA IS IN THE CORRECT DIR.
    if extract:
        os.makedirs('./data/raw_lidar/extracted_data', exist_ok=True)
        # dir = './data/raw_lidar'
        os.listdir(directory)
        extracted_lidar_dir = './data/raw_lidar/extracted_data'
        os.makedirs('./data/raw_lidar/extracted_data', exist_ok=True)

        for file in os.listdir(directory):
            # we have tar files. we can extract these with a lf function
            if file.endswith('.tar'):
                full_path_file = os.path.join(directory, file)
                print(f'extracting {full_path_file} to {extracted_lidar_dir}...')
                lf.extract_data(full_path_file, extracted_lidar_dir)

        # Tar files are files of files. This creates a messy directory. This will move it from that directory to make it a little easier to navigate
        # Define the source and destination paths
        temp_dir = './data/raw_lidar/extracted_data/Users/HISCOX/OneDrive - University of South Carolina/Research/SAVANT/SAVANT Data/Level 1 - Grouped/USC Aerosol Lidar/ASCII'
        source_directory = './data/raw_lidar/extracted_data/ASCII/'
        destination_directory = extracted_lidar_dir

        try:
```

```python
            # Move the directory
            for folder in os.listdir(source_directory):
                print(folder)
                shutil.move(os.path.join(source_directory,folder), destination_directory)
                print(f"Directory '{source_directory}' moved successfully to '{destination_directory}'")
        except FileNotFoundError:
            print(f"Error: Source directory '{source_directory}' not found.")
        except Exception as e:
            print(f"An error occurred: {e}")


def read_fastdata(file):
    '''
    Read high-frequency data from a netCDF file into a pandas dataframe.
    This is developed specifically for the SAVANT wind data I have available.
    This data is collected at 4-hour increments, with a sampling rate of 20 Hz (20 times per second).
    I am interested in wind and temperature data to decompose the heat flux.

    Parameters
    ----------

    Returns
    ----------

    Examples
    ----------

    '''

    with Dataset(file, 'r') as ds:
        time_units = ds.variables['time'].units[14:]
        time_1s = ds.variables['time'][:]
        num_intervals = ds.dimensions['time'].size   # should be 14400, for 4 hours in seconds
        samples_per_interval = ds.dimensions['sample'].size   # should be 20 for 20 hertz or 20 times per second
        total_samples = num_intervals * samples_per_interval

        delta_t = ds.variables['time'].__dict__['interval(sec)']/samples_per_interval

        start_time = time_1s[0]
        end_time = start_time + 14400

        time_20hz = np.arange(start_time, end_time, delta_t)
        print(len(time_20hz))

        # extract wind data
        data_dict = {}
        wind_fields = [var for var in ds.variables.keys() if hasattr(ds.variables[var], 'long_name')
                       and 'wind' in ds.variables[var].long_name.lower()]
        for field in wind_fields:
            if ds.variables[field].shape==(14400, 20):
                samples_2d=ds.variables[field][:]
                time_series_1d = samples_2d.flatten().data
                data_dict[field]= time_series_1d
            else:
                print(f'{field} not used. Incorrect length.')
        temperature_fields = [var for var in ds.variables.keys() if hasattr(ds.variables[var], 'long_name')
                              and 'temp' in ds.variables[var].long_name.lower()]
        for field in temperature_fields:
            if ds.variables[field].shape==(14400, 20):
                samples_2d=ds.variables[field][:]
                time_series_1d = samples_2d.flatten().data
```

```
                data_dict[field]= time_series_1d
            else:
                print(f'{field} not used. Incorrect length.')

    df = pd.DataFrame(data_dict, index=time_20hz)

    try:
        df.index = pd.to_datetime(df.index, unit='s', origin=time_units)
    except ValueError as e:
        print(f"Warning: Could not set datetime index due to error: {e}")
        # If conversion fails, keep the index as seconds (float)

    df.index.name = 'time'

    return df

def calc_wind(u, v):
    '''
    Calculates wind speed (m/s) and direction (0-360 degrees) given
    u (zonal) and v (meridional) components.

    Parameters
    ----------
        u (float, int): wind speed in zonal direction or x
        v (float, int): wind speed in meridional direction or y

    Returns
    ----------
        windspeed and direction

    Examples
    ----------
        spd, dir = calc_wind(3,4)
        print(spd, dir)


    '''
    speed = np.sqrt(u**2 + v**2)
    direction = (np.rad2deg(np.arctan2(u, v)) +360) %360
    return speed, direction

def create_winds(df):
    '''
    Adds wind speed and wind direction using the calc_wind function.

    Parameters
    ----------

    Returns
    ----------

    Examples
    ----------

    '''
    df = df.copy()
    height_suffixes = set()
    for col in df.columns:
        suffix = col[col.find('_'):]
        print(col, suffix)
        height_suffixes.add(suffix)

    for suffix in height_suffixes:
        u_col = 'u'+suffix
        v_col = 'v'+suffix
        if u_col in df.columns and v_col in df.columns:
            print(f'calculating {u_col} and {v_col}')
```

```python
            df[f'spd{suffix}'],  df[f'dir{suffix}'] = calc_wind(df[u_col], df[v_col])
    return df

def read_metadata(directory, file_path = None):
    '''
    Reads metadata from Raymetrics lidar .txt files.

    Parameters
    ----------
        directory (string): directory of lidar txt files.
        file_path (string, optional): path to lidar file. if None, the entire directory will be read.

    Returns
    ----------
        Pandas DataFrame with metadata from txt file(s) processed.

    Examples
    ----------

    '''

    if file_path == None:
        files_to_read = [txt for txt in os.listdir(directory) if txt.endswith('.txt')]
    if file_path != None:
        files_to_read = [file_path]

    meta_df = pd.DataFrame(columns=['file', 'start', 'end', 'elevation', 'longitude', 'latitude', 'zenith', 'azimuth', 'temp_ground', 'pressure'])

    # Convert appropriate columns to numeric
    numeric_cols = ['elevation', 'longitude', 'latitude', 'zenith', 'azimuth', 'temp_ground', 'pressure']
    # meta_df[numeric_cols] = meta_df[numeric_cols].apply(pd.to_numeric, errors='coerce')  # Convert and set non-numeric values to NaN

    for txt_file in files_to_read:
        f = os.path.join(directory, txt_file)
        meta_data = []
        shot_data = []
        with open(f, 'r') as file:
            content = file.readlines()

            # The first seven lines are meta data
            meta = content[0:7]

            header1 = meta[0]
            header2 = meta[1]
            header3 = meta[2]
            header4 = meta[3]
            header5 = meta[4]
            header6 = meta[5]
            header7 = meta[6]

            header2 = header2.split(' ')
            campaign = header2[0]
            startDate = f'{header2[3]} {header2[4]}'
            endDate = f'{header2[5]} {header2[6]}'
            elevation = float(header2[7])
            longitude = float(header2[8])
            latitude = float(header2[9])
            zenith = float(header2[10])*-1
            azimuth = float(header2[11])
            temp_ground = float(header2[12])
            pressure = float(header2[13])
            bin_width = float(header5.split()[6])
            data = content[7:]
```

```python
        # Create a dataframe for the meta data
        meta_data.append([f, startDate, endDate, elevation, longitude, latitude, zenith, azimuth, temp_ground, pressure])

        # Create DF
        temp_meta_df = pd.DataFrame(meta_data, columns=['file', 'start', 'end', 'elevation', 'longitude', 'latitude', 'zenith', 'azimuth', 'temp_ground', 'pressure'])

        # Convert appropriate columns to numeric
        # numeric_cols = ['elevation', 'longitude', 'latitude', 'zenith', 'azimuth', 'temp_ground', 'pressure']
        # temp_meta_df[numeric_cols] = temp_meta_df[numeric_cols].apply(pd.to_numeric, errors='coerce')  # Convert and set non-numeric values to NaN

        # Convert date columns to datetime
        temp_meta_df['start'] = pd.to_datetime(temp_meta_df['start'], errors='coerce', dayfirst=True)
        temp_meta_df['end'] = pd.to_datetime(temp_meta_df['end'], errors='coerce', dayfirst=True)
        meta_df = pd.concat([meta_df, temp_meta_df], ignore_index=True)
    meta_df = meta_df.sort_values('start')
    return meta_df

def find_scans(df, start_time=None, end_time=None):
    '''
    '''
    # find local min max indices
    max_zenith_idxes= find_peaks(df['zenith'].values)[0] #argrelextrema(df['zenith'].values, np.greater)[0]
    min_zenith_idxes = find_peaks(-df['zenith'].values)[0] # argrelextrema(df['zenith'].values, np.less)[0]

    # Merge and sort extrema indices
    min_max = np.sort(np.concatenate((max_zenith_idxes, min_zenith_idxes)))
    min_max = np.insert(min_max, 0, 0)   # insert 0 so we can start with the initial row.
    if len(df['zenith'])-1 not in min_max:   # add last row to the end.
        min_max = np.insert(min_max, len(min_max), len(df['zenith'])-1)

    # organize lists of indexes
    times = []
    zeniths = []
    for extrema in min_max:
        row = df.iloc[extrema]
        times.append(row['start'])
        zeniths.append(row['zenith'])

    start_stop_pairs = []
    for i in range(len(min_max)-1):
        if min_max[i+1] - min_max[i] > 4:
            start_stop_pairs.append([int(min_max[i]), int(min_max[i+1])])

    time_stamp_pairs = []
    for pair in start_stop_pairs:
        time_stamp_pairs.append([df.iloc[pair[0]]['start'], df.iloc[pair[1]]['start']])


    # Plot a figure to test.
    plt.figure(figsize=(18,8))
    plt.plot(df['start'], df['zenith'])
    plt.title(f"{df.iloc[0]['start']}-{df.iloc[-1]['start']}")

    plt.scatter(df['start'], df['zenith'], s=10)
    plt.scatter(times, zeniths, color='red')

    if start_time != None and end_time != None:
```

```python
        t1 = pd.to_datetime(start_time)
        t2 = pd.to_datetime(end_time)
        plt.xlim(t1, t2)
    for row in start_stop_pairs:
        plt.axvspan(df.iloc[row[0]]['start'], df.iloc[row[1]]['start'], color='grey', alpha=0.25)
    plt.show()
    return start_stop_pairs, time_stamp_pairs


def file_to_data(txt_file, max_distance=400, bin_width=7.5, origin_X=0, origin_Y=0, origin_Z=2, origin_azimuth=0, origin_zenith=0, bg_len_cutoff=4, filter_t=2):
    '''
    Reads a LiDAR text file, extracts measurement data, processes background noise correction, and computes
    spatial coordinates for each data point.

    Parameters
    ----------
        txt_file (str): Path to the LiDAR text file.
        distance (float, optional): Maximum measurement distance. Default is 450.
        bin_width (float, optional): Width of each measurement bin. Default is 7.5.
        origin_X (float, optional):
        origin_Y (float, optional):
        origin_Z (float, optional):
        origin_azimuth (float, optional):
        origin_zenith (float, optional):
        filter_t (float, optional): threshold filter. Default set to 1 * sigma

    Returns
    ----------
       pandas.DataFrame: A DataFrame containing the processed LiDAR data with background noise correction and
            spatial coordinates (x, z, distance).

    Examples
    ----------

    '''
    with open(txt_file, 'r') as file:
        content = file.readlines()

    meta = content[:7]
    shot_data = meta[1]
    Location, StartDate, StartTime, EndDate, EndTime, Elevation, Longitude, Latitude, Zenith, Azimuth, Temp, Pressure = shot_data.split()
    Zenith = -float(Zenith)
    last_row = int(math.ceil(max_distance/bin_width))
    data = [row.strip().split('\t') for row in content[7:(7+last_row)]]

    data = pd.DataFrame(data, columns=['analog', 'photon']).apply(pd.to_numeric)

    # Add data from meta data
    data['start'] = pd.to_datetime(f"{StartDate} {StartTime}", format='%d/%m/%Y %H:%M:%S')
    data['end'] = pd.to_datetime(f"{EndDate} {EndTime}", format='%d/%m/%Y %H:%M:%S')
    data['zenith'] = Zenith

    # Add distance information
    step_size = bin_width
    data['distance'] = (data.index+1) * step_size
    data['x'] = data['distance'] * np.cos(np.radians(Zenith))
    data['z'] = origin_Z + data['distance'] * np.sin(np.radians(Zenith))

    # First perform Background correction. We need to remove the noise.
    bg_length = min(1000, int(len(data)/bg_len_cutoff))    # bg_length = min(1000, int(len(data)/4))
```

```python
    EndSig_A, EndSig_P = data['analog'][-bg_length:].mean(), data['photon'][-bg_length:].mean()
    EndSig_A_std, EndSig_P_std = data['analog'][-bg_length:].std(), data['photon'][-bg_length:].std()

    data['analog_bgc'] = np.where(data['analog'] >= EndSig_A + filter_t * EndSig_A_std, data['analog'] - EndSig_A, 0) # old:---> data['analog_bgc'] = np.where(data['analog'] >= EndSig_A + 3 * EndSig_A_std, data['analog'] - EndSig_A, 0)
    data['photon_bgc'] = np.where(data['photon'] >= EndSig_P + filter_t * EndSig_P_std, data['photon'] - EndSig_P, 0) # old:---> data['photon_bgc'] = np.where(data['photon'] >= EndSig_P + 3 * EndSig_P_std, data['photon'] - EndSig_P, 0)

    # Range correction. Inverse-square law of light.
    Ranges=data['distance'].values
    data['analog_rcs']=data['analog_bgc'] *Ranges**2
    data['photon_rcs']=data['photon_bgc'] *Ranges**2

    # Normalize data
    peak=max(data['analog_rcs'])
    data['analog_rcs_norm']=data['analog_rcs']/peak

    return data

def process_iop(directory, max_distance=400, filter_t=2):
    '''
    Processes the entire directory using the file_to_data function.

    Parameters
    ----------
        directory (str): Path to the LiDAR text file.

    Returns
    ----------
       pandas.DataFrame: A DataFrame containing the processed LiDAR data with background noise correction and
            spatial coordinates (x, z, distance) for the entire directory.

    Examples
    ----------

    '''
    df = pd.DataFrame()
    for file in os.listdir(directory):
        if file.endswith('.txt'):
            temp_df = file_to_data(os.path.join(directory, file), max_distance=max_distance, filter_t=filter_t)
            df = pd.concat((df, temp_df), ignore_index=True)
    return df

def plot_contour_scan(scan_df, column="analog_rcs", title=None, x_limits=None, y_limits=None, method='linear', surface=None, mark_max=False):
    x = scan_df["x"]
    z = scan_df["z"]
    value = scan_df[column]

    # Create a grid
    xi = np.linspace(x.min(), x.max(), 100)
    zi = np.linspace(z.min(), z.max(), 100)
    Xi, Zi = np.meshgrid(xi, zi)

    # Interpolate data
    Ai = griddata((x, z), value, (Xi, Zi), method=method) # methods: linear, nearest, cubic

    # Plot contour map
    plt.figure(figsize=(15, 6))
    contour = plt.contourf(Xi, Zi, Ai, cmap="turbo", levels=30) # gist_ncar
```

```
        plt.colorbar(label="Relative Backscatter")
        plt.xlabel("Distance (m)")
        plt.ylabel("Height (m)")
        if title == None:
            plt.title("Contour Map of Normalized Backscattter")
        else:
            plt.title(title)
        if surface:
            plt.plot(surface['x'], surface['z'])
        if x_limits:
            plt.xlim(x_limits[0], x_limits[1])
        if y_limits:
            plt.ylim(y_limits[0], y_limits[1])


        if mark_max==True:
            # find point of max
            max_rcs_index = scan_df['analog_rcs'].idxmax()
            max_backscatter_data = scan_df.loc[max_rcs_index]

            max_x = max_backscatter_data['x']
            max_z = max_backscatter_data['z']

            plt.annotate(
                'Maximum Backscatter',
                xy=(max_x, max_z),
                xytext=(100,25),
                arrowprops=dict(
                    arrowstyle='->',
                    color='red',
                    lw=2
                ),
                fontsize=12,
                color='green'
            )

        plt.show()
```