Background

Acoustic Dopper Velocimeters (ADVs) are standard oceanographic instruments for measuring water velocity at high frequency, usually between 4 - 100 Hz. These measurements allow researchers to study turbulence in the ocean, which is important for understanding ocean physics and the ocean's role in the Earth's climate.

ADVs work by sending sound waves into the water and then listening to the return echo after the sound waves bounce off of suspended particles in the water. By using the Doppler shift between subsequent pulses (i.e., the change in frequency of the returned sound due to the particle moving with the flow), ADVs can estimate the velocity of the water. This principle is illustrated in Figure 1.

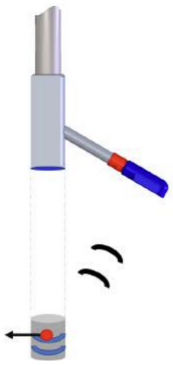

Figure 1: Pair of transmit pulses (blue), moving particle (red), and Doppler-shifted return echos (black). Image courtesy of Nortek, Inc.

One problem with ADV data is that it is often very noisy/spiky, i.e., characterized by large fluctuations in estimated velocity that do not really represent the fluid velocity. This is mostly due to inherent limitations in using the Doppler effect to measure velocity, but could also be caused by reflections of the sound waves off of other stuff in the water (the sea floor, fish, etc.). As a result oceanographers have proposed numerous methods to "de-spike" ADV data. In this assignment, you will implement the most popular method, which was proposed by Goring & Nikora (2002).

Assignment

In the provided `adv.csv` you will find a 14-minute sample of ADV data that I collected in South San Francisco Bay in July, 2018. The time column is in units of seconds starting at 0. The U column provides the water speed in m/s (we won't worry about flow direction in this assignment). The goal of this assignment is to clean the ADV data while practicing:

- Object-oriented programming
- Data visualization
- Outlier identification
- Statistics and analytic geometry
- Programming flow control

You should complete your assignment in the `despike_adv.ipynb` Jupyter Notebook in the Github repository that is created for you when you accept the assignment. You should intersperse text, code, and plots so that it tells a nice story about how you cleaned the data. If you need a reminder on how Classes work in Python, see the docs here and refer to the Lecture 1 Intro To Python notebook.

Part 1

To begin, we will try to clean up the ADV data using visual techniques. To that end you should do the following:

1. Define a class named `AdvData` that can be initialized with a path to a data file. The initialization should also define attributes `self.u` and `self.t` that store the velocity and time data as numpy arrays.
2. Define methods on the class that make a time-series plot and histogram of the velocity. Produce plots using each of these methods, and comment on whether you can distinguish between the noise and the real signal using these visual tools.
3. Define another method that applies a user-specified condition or threshold to the data which sets any "bad" data points equal to `np.nan` and

saves the modified array to `self.u_filtered_simple`. When percentage of the data does this threshold set to `NaN`?

4. Make a plot showing the original signal and your filtered signal (with the condition applied) on top of it in a different color. Comment on the effectiveness of this method.

To Turn In: Answer questions 1-4 (including all code, plots, and comments) in the `despike_adv.ipynb` Jupyter Notebook.

Part 2

Now we will implement Goring & Nikora's algorithm. For background, you should read the Introduction and Methods section of the paper up until the "Algorithms" heading. The basic idea is that we are going to draw ellipses in three different coordinate systems, where the axes are different combinations of the velocity and its first two time derivatives. Data points inside the ellipses we draw are ok, but data points that fall outside the ellipses are bad, and need to be interpolated over.

Once you read the background information, you can then follow the Phase-Space Thresholding section of their original paper, but I find it a little confusing, so I rewrote the instructions below for clarity. Please implement the algorithm as a method on the `AdvData` class, so that it results in a new attribute `self.u_filtered_gn` containing the filtered velocity array.

Algorithm Implementation:

1. Calculate estimates of the first and second derivative of the velocity

$$\Delta u_i = (u_{i+1} - u_{i-1})/2 \tag{1}$$
$$\Delta^2 u_i = (\Delta u_{i+1} - \Delta u_{i-1})/2 \tag{2}$$

   where the subscript i denotes the i^{th} element of the vector u. You may find the `np.gradient` function helpful here.

2. Calculate the standard deviation of velocity and its first two derivatives, σ_u, $\sigma_{\Delta u}$, and $\sigma_{\Delta^2 u}$.

3. Calculate the universal maximum scaling factor $\lambda = \sqrt{2 \ln n}$, where n is the number of velocity observations. You can think of this as the

maximum number of standard deviations of variability you can expect to observe for a Gaussian process sampled n times.

4. Calculate the rotation angle θ for the u-$\Delta^2 u$ ellipse as

$$\theta = \tan^{-1}\left(\frac{\sum u_i \Delta^2 u_i}{\sum u_i^2}\right) \tag{3}$$

5. Calculate the major and minor axes of each of the three ellipses as follows:

   (a) u-Δu: define the major axis $a_1 = \lambda\sigma_u$, and the minor axis $b_1 = \lambda\sigma_{\Delta u}$

   (b) u-$\Delta^2 u$: Calculate the major and minor axes a_2 and b_2 as the solutions to the following set of equations:

$$(\lambda\sigma_u)^2 = a_2^2 \cos^2\theta + b_2^2 \sin^2\theta \tag{4}$$
$$(\lambda\sigma_{\Delta^2 u})^2 = a_2^2 \sin^2\theta + b_2^2 \cos^2\theta \tag{5}$$

   I would suggest treating a_2^2 and b_2^2 as the unknowns, solving the resulting set of 2 equations with 2 unknowns, and then taking the square roots of the answers to find a_2 and b_2.

   (c) Δu-$\Delta^2 u$: define the major axis $a_3 = \lambda\sigma_{\Delta u}$ and the minor axis $b_3 = \lambda\sigma_{\Delta^2 u}$

6. Find the union of all "bad" indices where any of the pairs $(u, \Delta u)$, $(u, \Delta^2 u)$, or $(\Delta u, \Delta^2 u)$ fall outside of the bounds of their respective ellipses (e.g., Figure 4b in Goring & Nikora). This corresponds to any data pair (x, y) that satisfies the condition:

$$\frac{(\cos\theta(x-\overline{x}) + \sin\theta(y-\overline{y}))^2}{a^2} + \frac{(\sin\theta(x-\overline{x}) - \cos\theta(y-\overline{y}))^2}{b^2} > 1$$

   where x is the major axis variable, y is the minor axis variable, θ is the rotation of the major axis, and a bar $\overline{()}$ denotes the average value.

   (a) For the u-Δu ellipse, u is on the major axis, Δu is on the minor axis, and the ellipse is not rotated ($\theta = 0$)

   (b) For the u-$\Delta^2 u$ ellipse, u is on the major axis, $\Delta^2 u$ is on the minor axis, and the ellipse is rotated by θ

(c) For the Δu-$\Delta^2 u$ ellipse, Δu is on the major axis, $\Delta^2 u$ is on the minor axis, and the ellipse is not rotated ($\theta = 0$).

7. Set velocity to `np.nan` at any of the bad indices, and then impute over the `NaN` value using a method of your choosing (e.g., linear interpolation, nearest neighbor interpolation, replacement by a moving average value, etc.).

8. Repeat steps 1 - 7 until the total number of bad indices is less than 5.

At each iteration of the algorithm, you should output a plot like Figure 2 showing the data points with the associated ellipses. This will allow you to track the progress of the algorithm after each bad data replacement step. You may find it easier to plot the ellipses using the parametric equation for a (potentially rotated) ellipse, which you can look up.

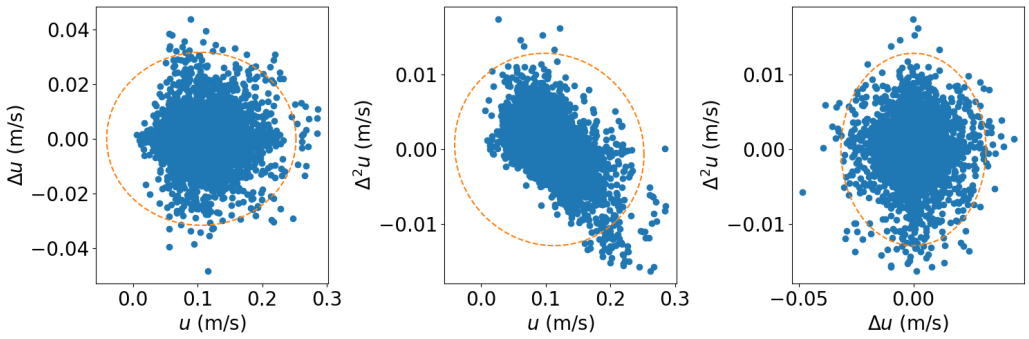

Figure 2: Snapshot of the despiking aglorithm progress after a certain number of iterations. Markers denote the data points, some of which were originally bad and have been replaced, while the dashed line separates the good data points from remaining bad data points.

Hint: If at any point you need to make a copy of a numpy array, it is better to do something like `copied_array = original_array.copy()` instead of `copied_array = original_array`. In the latter case, you might accidentally modify `original_array` when you modify `copied_array`. In the former case where you use `.copy()`, there is no danger of that happening.

To Turn In: Add your implementation of the despiking algorithm to the `despike_adv.ipynb` Jupyter Notebook as a method of the `AdvData` class. Please also address the following:

- How many iterations did your algorithm take to converge?
- Make a plot showing the original signal and your filtered signal (using the Goring & Nikora algorithm) on top of it in a different color. Comment on the effectiveness of this method compared to the visual outlier detection you implemented in Part 1.