SWE Challenge - Quantum Diamonds

Saurabh Pandey

Contents

1. Challenge

So the challenge will be for you to work with the fundamental underlying data of QD. The data is shown in the attached paper in Figure 1.(d)[1]. I also included an esr.py file which visualizes a similar spectrum and makes it easy to get started.
The goal here is to extract 3 values: The peak differences between peak cluster 1 & 6, peak cluster 2 & 5, peak cluster 3 & 4.

- These peak clusters are composed of 3 peaks each and can be mapped using a triple lorentzian distribution. What we are interested in is the center of a triple which is the average of the 3.
- For reference: Peak cluster means a single graph that is shown within a dashed black frame in Figure 1.(d) [1].
- For reference: There are two measurements in the directory, the one is a reference measurement with no external magnetic field applied, for the other one there was a 45 mA current applied in the object that was measured which creates an external magnetic field. Both are relevant to the overall results.

These 3 peak deltas allow us to **calculate the magnetic field strengths in (x,y,z) direction**.

2. Approach

The basic pipeline used to solve the challenge is depicted in fig. 1. In short, once the data is smoothed, and clipped, peaks(i.e. dips in our case) are found for each segment (i.e. step interval of the dataset). The obtained peaks are then filtered to obtain the final three peaks and the calculations are done accordingly. From this onwards, the data without an external magnetic field will be called an Idle wave, and one with will be called an active wave.

3. Solution

3.1. EDA

Some statistics about the provided data:

- Shape of X data = (906,)
- Shape of Y data = (2, 906, 1, 50, 50)
- Steps interval = 151

The Y data for both idle and active waves is multidimensional with two 906 samples, where each sample is (50,50) dimensional. When these (50,50) arrays are visualized, there is no visible difference for the same wave i.e. The two copies of (906, 50, 50) samples for the idle wave are visually indistinguishable(generally) same for the active wave. This is further supported by the statistics :

Table 1. Statistics for Dimension 1 and 2 for both type of waves

Type	Mean Dim 1	Mean Dim 2	Std Dim 1	Std Dim 2
Idle	14381	14399	6571	6576
Active	14976	14995	6874	6879

Note: Reported numbers are rounded to nearest integer

But when idle and active waves are compared there is a significant visual difference i.e. active wave have bigger intensity values less distributed as shown in fig. 2

3.2. Normalisation

```python
def normalise_dataset(self, data):
    return np.sum(data[0] / data[1], axis=1)
```

Code 1. Normalisation

The above code is used to normalize the data. Given that the information in the first 2 dimensions is almost the same. I could not figure out the reason for two similar readings for each wave that is used in the above code to do a kind of normalization. Either these are two separate readings of the same area under the same conditions to have robust readings(i.e. some sort of SNR management) or with constant readings that are known to scale the values in a range. (The best I can do at this point is guess).

3.3. Filtering

```python
def smooth_data(self, data, window_length=15,
 poly_order=3):
    return savgol_filter(data, window_length,
 poly_order)
```

Code 2. Filtering

As you can see in fig. 3,4, data is noisy and peaks cluster are difficult to figure out. To reduce the noise we need to use a proper filter that suits the use case i.e. The effect on the peaks is least. One of the popular filters for spectral data is **Savitzky-Golay filter**. The fig. 5, 6 plot the smooth data for both the waves. Of course, it's obvious to ask why not a popular filter like Gaussian, but as depicted in fig. 7 Gaussian affects the shapes of the peaks comparatively more than Savitzky-Golay. One of the side effects of smoothing is the loss of accurate y-axis(intensity information due to reduced peak sizes).

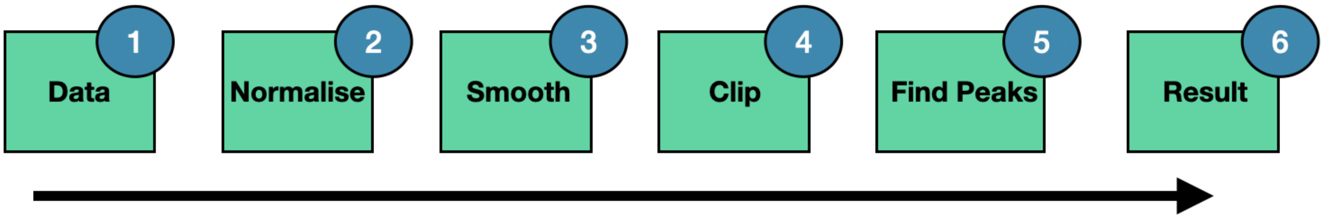


Figure 1. Pipeline for processing the dataset

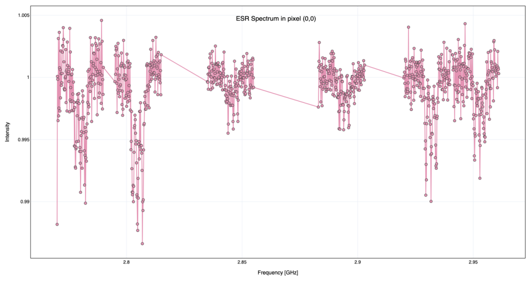


Figure 3. Visualisation of Idle wave

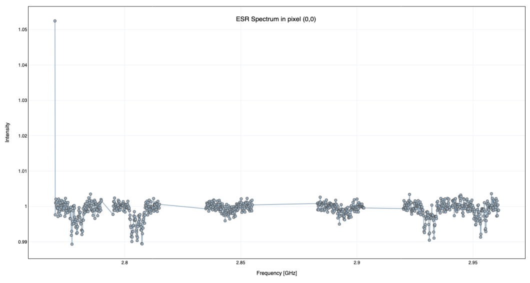
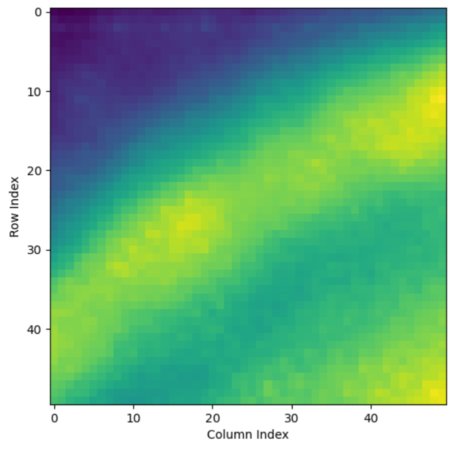
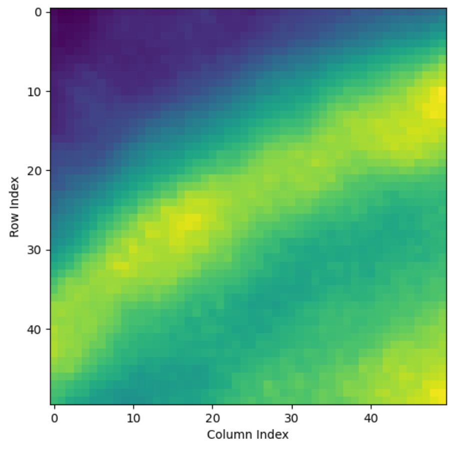


Figure 4. Visualisation of Active wave



(a) Idle Wave



(b) Active Wave

Figure 2. Visualisation of one (50,50) sample

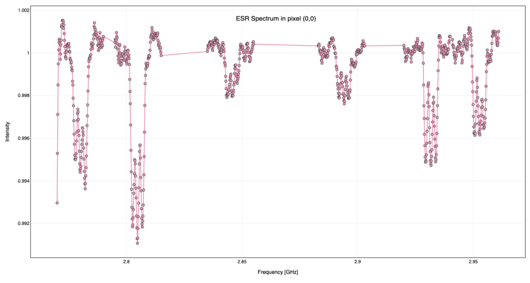


Figure 5. Smooth Idle wave

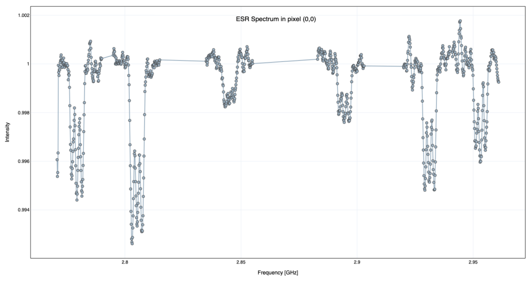


Figure 6. Smooth Active wave (some points are clipped for better visualization)

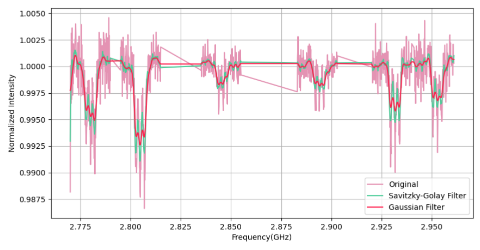


Figure 7. Comparison Savitzky-Golay vs Gaussian

3.4. Outliers

As visible from fig. 5,6 there are tails(can be considered as outliers) in both waves.

```python
def get_clip_range(self, step_interval_list, clip_percentage=.10):
    return int(step_interval_list[0] * clip_percentage)
```

Code 3. Clipping

I use the above code to find the number of points based on the clipping percentage to remove from both ends of the waves, that middle segment values are not touched. This of course is not a very sound statistical way at the same time a common approach like IQR(interquartile range) based removal will affect the actual peak values, even if the removal is applied on a small percentage of tail-end points, the approach boils down to finding the right percentage, which a conservative manual clipping approached used here is doing anyways. Also, as the percentage is used to calculate the number of points to clip from both ends of the wave, it's relative to the number of points in the interval. Alternatively, a better statistical approach can be found if all this clipping is desired(the task can be solved even without clipping step).

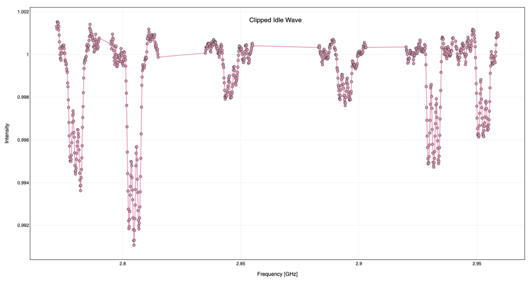


Figure 8. Clipped 10% values from endpoints of Idle wave

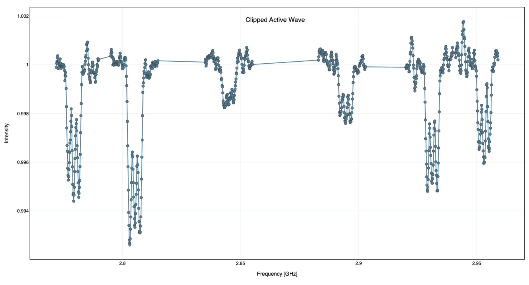


Figure 9. Clipped 10% values from endpoints of Active wave

3.5. Fitting

Now the approach is very straight forward.

- Each wave is made of of 6 segments. Few segments are visualised in fig. 10-12
- The length of these segments are given by step intervals
- Loop over each segment and find peaks in that segment
- Using a function like find_peaks from scipy library will give you alot of peaks(unless a lot manual parameter tuning is done which might not generalise well).
- Once you have a lot of peaks, the lowest three peak values are the ones we are looking for. Lowest because we are looking for dips not peaks.

```python
def chunk_array_by_sizes(self, data, chunk_sizes):
    chunks = []
    start_index = 0

    for size in chunk_sizes:
        if start_index + size > len(data):
            raise ValueError("Chunk sizes exceed the length of the data.")
        end_index = start_index + size
        chunk = data[start_index:end_index]
        chunks.append(chunk)
        start_index = end_index

    return chunks
```

Code 4. Chunking

The above code takes the data and chunks it into 6 segments of appropriate length, followed by peak detection.

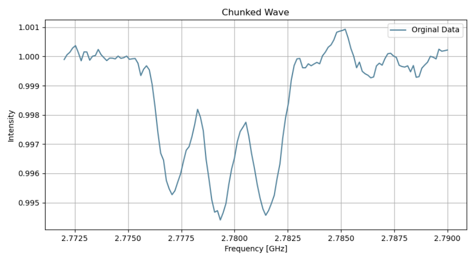


Figure 10. Chunked Wave (Cluster - I)

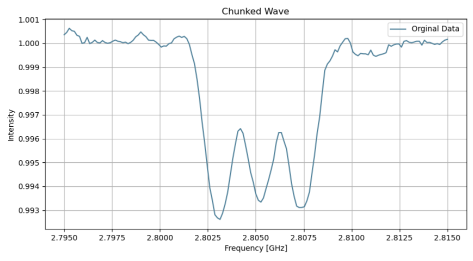


Figure 11. Chunked Wave (Cluster - II)

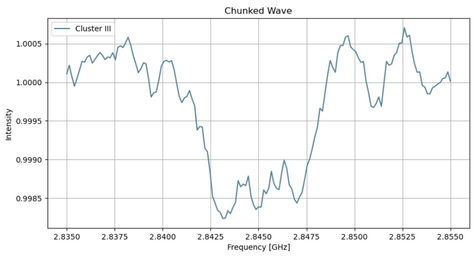


Figure 12. Chunked Wave (Cluster - III)

3.6. Peak Detection

Below code is used to detect peaks for each segment. The first if condition is executed as we are looking dips. For dips we just need to negate the data converting dips to peaks.

```python
def get_peaks(self, data, distance=5, dips=True):
    if dips:
        return find_peaks(-data, distance=distance)
    else:
        return find_peaks(data, distance=distance)
```

Code 5. Peak detection

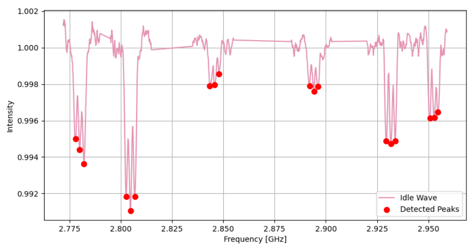


Figure 13. Idle Wave Detected Peaks

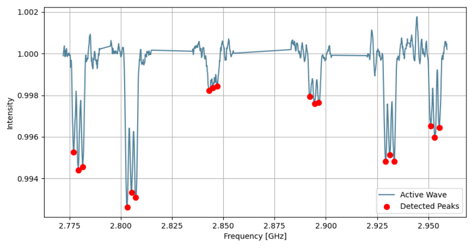


Figure 14. Active Wave Detected Peaks

3.7. Calculation

The below code calculates the peak centers and difference between active and idle waves

```python
def get_peaks_delta(self, peaks, frq):
    delta_list = []
    delta_one_six_peak = np.abs((np.sum(frq[peaks[0:3]]) / 3) - (np.sum(frq[peaks[15:18]]) / 3))
    delta_two_five_peak = np.abs((np.sum(frq[peaks[3:6]]) / 3) - (np.sum(frq[peaks[12:15]]) / 3))
    delta_three_four_peak = np.abs((np.sum(frq[peaks[6:9]]) / 3) - (np.sum(frq[peaks[9:12]]) / 3))
    delta_list.extend([delta_one_six_peak, delta_two_five_peak, delta_three_four_peak])
    return delta_list


def print_results(self, peaks_idle, frq_idle, peaks_active, frq_active):
    delta_f_idle_list = self.get_peaks_delta(peaks_idle, frq_idle)
    delta_f_active_list = self.get_peaks_delta(peaks_active, frq_active)
    print(f"Difference Between 1-6 Peak:{(delta_f_active_list[0] - delta_f_idle_list[0]) * 1e-9}")
    print(f"Difference Between 2-5 Peak:{(delta_f_active_list[1] - delta_f_idle_list[1]) * 1e-9}")
    print(f"Difference Between 3-4 Peak:{(delta_f_active_list[2] - delta_f_idle_list[2]) * 1e-9}")
```

Code 6. Calculation

Table 2. Results for peak deltas between Active and Idle wave

Peak Cluster	Value(GHz)
1-6	0.001156
2-5	-0.000845
3-4	0.000889

3.8. Fitting Lorentzian

All though the above method works, it is prone to very noisy and outlier prone data. One false peak can impact the results heavily. A better approach is to fit a curve, specifically an inverted lorentzian. For a good fit at this step, the peak detection step is crucial for initial guess of the peak locations, their amplitudes, width and center.

```python
def lorentzian(self, x, amp, cen, wid):
    return -amp * wid ** 2 / ((x - cen) ** 2 + wid ** 2)

def fit_all_clusters(self, x, *params):
    assert len(params) == 18 * 3
    result = np.zeros_like(x)
    for i in range(0, len(params), 3):
        amp, cen, wid = params[i:i + 3]
        result += self.lorentzian(x, amp, cen, wid)
    return result

def generate_parameters_for_fitting(self, x, y, peaks):
    parameter_list = []
    for index in range(0, len(peaks), 3):
        width = (x[peaks[index + 2]] - x[peaks[index]]) / 2
        peak_1 = [y[peaks[index]], x[peaks[index]], width]
        peak_2 = [y[peaks[index + 1]], x[peaks[index + 1]], width]
        peak_3 = [y[peaks[index + 2]], x[peaks[index + 2]], width]
        parameter_list.extend(peak_1)
        parameter_list.extend(peak_2)
        parameter_list.extend(peak_3)
    return parameter_list

def curve_fitting(self, x, y, peaks, depth):
    initial_guesses = self.generate_parameters_for_fitting(peaks)
    popt, pcov = curve_fit(self.fit_all_clusters, x, y, p0=initial_guesses,
                           maxfev=depth)
```

Code 7. Curve Fitting

The above code generates parameter estimates based on the detected peaks and fits a curve.

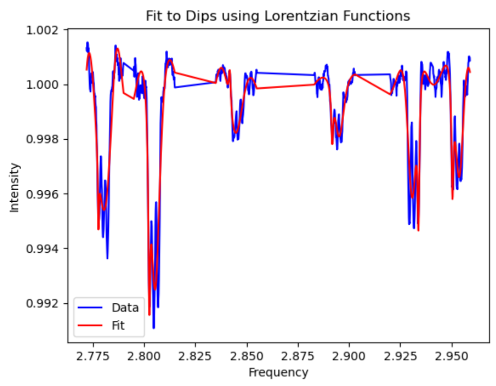


Figure 15. Triple Lorentzian Curve Fitting

The fit is not really good and needs better adjustment of width parameter, unfortunately I could not explore it any further. Although this approach is more robust, it is important to select the maxfev parameter big enough for successfully fitting, piece-wise fitting can circumvent this issue.

4. Code Structure

4.1. PeakAnalyzier Class

This class contains all the necessary functions required for the task. I have encapsulated most of the general purpose functions here, instead of individual functions. This class based approach is inline with modern frameworks like Django where class based views are more favoured.

4.2. Driver

The driver.py script implements the custom logic to solve the challenge using PeakAnalyzier class.

4.3. Tests

Small tests are provided as a placeholder to present the idea of unit tests. In no way these test are comprehensive or have any coverage. Mock based tests will be better than the direct ones I have provided.

4.4. Execute

- Create a virtual environment
- Activate the new virtual environment
- Execute the command $ pip install -r requirements.txt
- Execute the command $ python driver.py
- To run the tests $python -m python -m unittest

5. Future Works

5.1. Execution Time

Due to the independent nature of the sub problems (i.e. processing for each pixel), they can be executed in parallel. On a M1 Macbook air with 16GB RAM the execution for one pixel including the idle wave is 0.906 seconds on average. It would take more than 30 minutes to process the 50*50 pixels. This can be significantly reduced by using a parallel processing pipeline with a tool like **Pyspark**. In my experience a 10X speed could be expected.

5.2. Machine Learning

Honestly, I could not think of anyway to solve this problem using ML. Although some parts(i.e. de-noising) can be done using ML. A supervision based network could also be explored using the spectral images but my primary intuition for this task is going more towards a closed form solution rather than a predictive one.