nmr-analyser

A package for the automated analysis of ^{1}H NMR data

Scientific Programming Report

CHEM0062

Department of Chemistry
University College London

March 27, 2025

Contents

CHEM0062 Report

1 Summary

This report presents the development of a command line Python program for the automated analysis of ^{1}H NMR spectra, motivated by the need to classify complex splitting patterns. A custom peak picking algorithm was implemented to filter noise, followed by experimentation with clustering methods, ultimately combining k-means for initial estimation and Ward's method for final grouping. Original algorithms were developed to calculate cluster multiplicities, including a recursive backtracking approach and a simulated splitting tree to predict and match splitting patterns. The program was tested against experimental data from the Human Metabolome Database (HMDB), and demonstrated strong performance in classifying both simple and complex multiplets. Documentation is provided, including installation and use instructions and examples. The report concludes with my reflections on the project, its future potential, and the development of my programming skills, particularly in applying data structures and algorithms.

2 Background

Proton nuclear magnetic resonance (^{1}H NMR) spectroscopy is widely used in organic chemistry to determine molecular structures and identify key functional groups. A ^{1}H NMR spectrum consists of peaks, each corresponding to hydrogen atoms (protons) in a distinct chemical environment. These environments depend on the electronic surroundings of each proton, such as neighbouring electronegative atoms or functional groups, which alter the local magnetic field. This makes it possible to determine the number of hydrogen environments in a molecule and infer the presence of specific functional groups based on their chemical shift.

Peaks in NMR spectra do not always appear as single lines. Instead, they can split due to spin-spin coupling, which happens when a proton interacts with neighbouring non-equivalent protons. The simplest case follows the $n+1$ rule, where a proton with n equivalent neighbours splits into $n+1$ peaks. For example, in ethanol (CH_3CH_2OH), the three methyl (CH_3) protons are adjacent to two methylene (CH_2) protons. This causes the CH_3 peak to split into a triplet. Meanwhile, the CH_2 protons experience the effect of three neighbouring CH_3 protons and split into a quartet. This pattern arises because each proton can align or oppose the external field, creating multiple possible spin states.

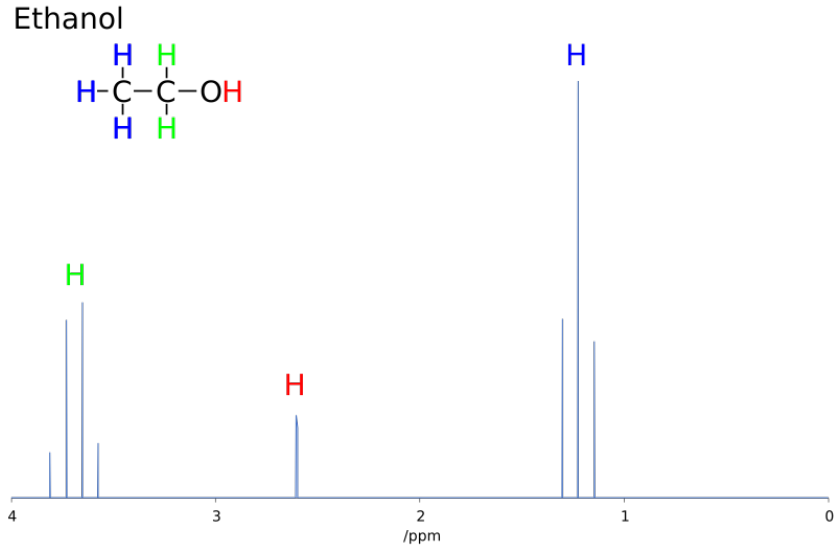


Figure 1: NMR spectrum of ethanol showing triplet and quartet splitting patterns due to spin-spin coupling. Figure from Andel.[1]

When a proton couples to two different sets of non-equivalent protons, the splitting pattern becomes more complex. A triplet-of-doublets (td) appears when a proton is split into a triplet by one set of two equivalent protons and then further split into a doublet by another single proton. For example, consider a proton coupled to two equivalent methylene protons (which produces a triplet) and another non-equivalent proton (which produces a doublet). This results in six peaks, with the spacing between them reflecting two different J-values. A doublet-of-triplets (dt) follows a similar logic but in the reverse order: the proton first splits into a doublet due to a stronger coupling, and then each peak splits again into a triplet by a weaker interaction. The order of splitting (whether a td or dt appears) depends on which coupling is stronger. The J-values reveal this—the larger J-value corresponds to the stronger coupling, which happens first in the splitting sequence.

The splitting tree diagram provides a useful way to visualise these interactions. Each level of the tree represents a step in the splitting process, with the distance between branches corresponding to the J-values. In a td splitting tree, the first division creates a triplet, followed by a smaller split into doublets. In a dt, the first split produces a doublet, and each peak is then further split into a triplet. By examining the spacing in a splitting tree, the relative strengths of different couplings can be determined, allowing the structural relationships between protons to be inferred.

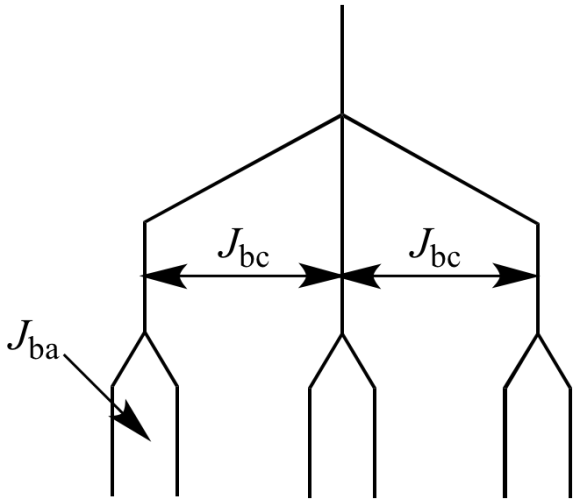


Figure 2: Splitting tree diagram illustrating the sequence of doublet and triplet splitting. Figure from Hardinger.[2]

Measuring J-values directly from an NMR spectrum is straightforward. The distance between the peaks of a multiplet (measured in ppm) is multiplied by the spectrometer frequency in MHz to give the coupling constant in hertz. For instance, if the peaks in a doublet are separated by 0.02 ppm on a 400 MHz spectrometer, the coupling constant is 8 Hz. Larger J-values suggest a stronger interaction, typically observed in trans-alkenes or geminal couplings, while smaller values are more common in cis-alkenes or long-range couplings.

However, assigning J-values and multiplicities manually can be complex. I have personally performed countless analyses in the lab, manually labelling peaks and their splitting patterns. While this process provides valuable structural insights, it is also time-consuming and prone to human error. Scientists typically inspect peak shapes, count peak separations, and estimate J-values by eye, making classification subjective and sometimes inconsistent. Even a basic automated classification would significantly speed up analysis and reduce mistakes, yet such automation is not commonly used when analyzing ^{1}H NMR data.

A number of tools already exist for automated peak picking. With libraries like SciPy, one can detect peaks in just a few lines of code, while packages such as *nmrglue* extend this functionality specifically to NMR data from various formats. However, the automated classification of multiplets remains challenging. One recent paper by Fischetti et al. demonstrates a deep learning approach that can handle complex patterns such as doublet-of-doublets, but it is not necessarily straightforward to integrate or customise.[3] I also experimented with NMRium, a web-based platform that attempts multiplet classification; in practice, it missed or mislabelled certain peaks in my lab data and offered no suggestions regarding possible functional groups.

Driven by these limitations, I decided to create my own program. It clusters peaks and predicts multiplicity using rigorous algorithms based on J-value spacing between peaks by mimicking the splitting tree. The program provides two output formats: a simple output with predicted multiplicities (e.g., doublet-of-doublets), and a detailed output listing all peak positions, J-values, and classifications. The detailed output allows users to troubleshoot incorrect predictions and

investigate discrepancies themselves. The goal is to offer a robust and efficient tool for ^{1}H NMR analysis.

3 Program specifications

Inputs

- A TSV (tab-separated values) file containing NMR spectral data, typically converted from Bruker or JCAMP datasets using NMRium or other software.
- The operating frequency of the spectrometer, entered manually as a command-line argument.
- Optional parameters including intensity threshold adjustments, uncertainty settings for J-value matching, and additional cluster testing.

Outputs

- A simple output file listing detected peaks along with their predicted multiplicities and possible functional groups.
- A detailed output file containing all peak positions, calculated J-values, and clustering information for troubleshooting and validation.
- A graphical output plotting the spectrum with detected peaks and their classifications.
- Direct terminal output displaying detected multiplets and J-values for quick review.

Details

The program begins by detecting peaks in the spectral data and prompts the user to adjust the Y-axis threshold if necessary. It also asks for confirmation on the number of detected peak clusters to ensure accuracy before proceeding. Once peaks are identified, the program generates all possible multiplicity patterns and predicts expected J-value intensities using splitting tree logic. These predicted intensities are then compared to the actual J-values extracted from the spectrum to determine the best matching multiplicity. This approach allows for the classification of complex peak structures, such as doublets of doublets or triplets of doublets.

4 Development

4.1 Data Input and Preparation

The project began with attempts to load Bruker data directly from the lab. However, converting this data into a Python-friendly format proved challenging, requiring extensive additional code, specialised libraries, and manual parsing. I also evaluated jcamp files, but these presented similar obstacles, particularly around data parsing and format consistency.

To simplify data handling, I switched to NMRium (https://www.nmrium.org/predict), a tool that converts Bruker datasets into TSV (tab-separated values) files. This change significantly improved data import and processing in Python. I began by generating a simpler, noise-free dataset within NMRium (`https://www.nmrium.org/predict?smiles=C1(C(C(OC(C1O)CO)OC)O)(O)`). This enabled early testing of the core functionalities such as data reading, peak detection, and plotting. These tests established a solid foundation for later stages of development, focusing on complex datasets.

4.2 Flowchart

The initial program design flowchart (Figure 3) presents the planned structure of the NMR analysis pipeline at the beginning of development. This flowchart was intended to guide the early stages of implementation, focusing on the core steps required to process and interpret spectral data. These included reading input files, detecting peaks using a threshold-based method, and performing preliminary clustering to group peaks into possible multiplets. At this point, the emphasis was on establishing a functioning framework for peak analysis, which could later be extended to include more detailed features such as multiplicity prediction.

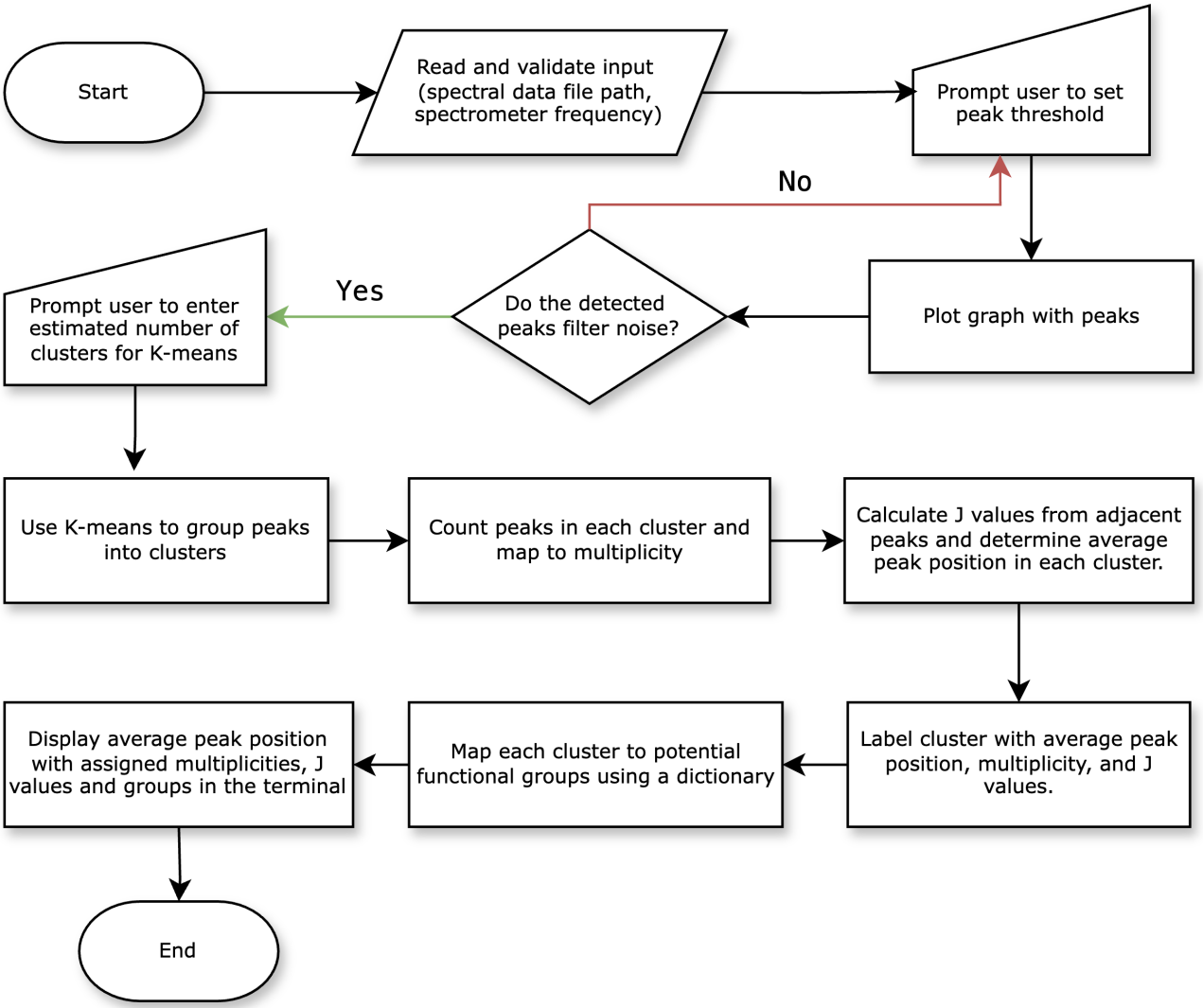


Figure 3: Initial program design flowchart outlining the core structure of the NMR analysis pipeline. Figure by the author.

4.3 Peak Picking and Smoothing Challenges

With the data in TSV format, I initially experimented with the `scipy.signal.find_peaks` function for peak detection. While this method offered a straightforward approach, I needed a more sensitive and customisable algorithm to accurately detect fine structures, such as triplet-of-triplets or doublet-of-doublets. Specifically, I wanted the algorithm to be highly sensitive by ensuring that a peak is only identified if it is higher than both its immediate neighbours. To achieve this precision, I developed a custom peak-picking algorithm that allowed users to set an intensity threshold and apply specific rules for peak identification. The algorithm labelled any data point exceeding the intensity threshold and meeting the strict peak criteria. The core logic of the custom algorithm is shown in the code snippet below:

Listing 1: Peak Picking Algorithm

```
def find_peaks(y, height):
    peaks = [i for i in range(1, len(y) - 1)
             if y[i] > height and y[i] > y[i - 1] and y[i] > y[i + 1]]
    return peaks
```

The custom method also included a `matplotlib` plot that visually displayed the detected peaks along with the threshold. The x-axis was automatically scaled to the dataset's minimum and maximum values, providing a clear visual guide that enabled users to quickly validate or adjust the threshold as needed.

During the development of the peak-picking algorithm, I tested several smoothing techniques to manage noise in the data. The goal was to reduce noise while preserving important peak structures, but this balance was difficult to achieve. I tested Gaussian, Wiener, and Savitzky-Golay filters. While these methods effectively reduced noise, they struggled with preserving smaller peaks and intricate multiplicities. In particular, these smoothing techniques often merged closely spaced peaks—such as those in a doublet of doublets—into a single broader feature. Since maintaining these fine distinctions is critical in proton NMR, I decided to remove the smoothing feature in later versions.

Although more advanced and specialised smoothing techniques for NMR data exist in the current literature, they were beyond the scope of this project.[4] However, the program was designed with a modular architecture, allowing future integration of these advanced methods if needed. This approach ensures that researchers or users can easily add and test more sophisticated smoothing algorithms within the existing framework.

4.4 Clustering Peaks into Multiplets

After establishing a reliable peak-picking method, the next step was to cluster the detected peaks into multiplets (e.g., singlet, doublet, triplet). Initially, I used k-means clustering to group peaks based on their x-values (ppm positions). While this method effectively clustered peaks, it required the user to manually estimate the number of multiplets beforehand, which was often difficult and dataset-dependent.

To automate the estimation of the number of clusters, I first experimented with Otsu's algorithm. Originally developed for converting grayscale images to black-and-white by clustering pixel values into two groups, Otsu's method can also be applied to the distances between detected peaks. The goal was to distinguish "small" gaps (within a multiplet) from "large" gaps (between multiplets). However, Otsu's method often proved too sensitive to outliers, resulting in thresholds that did not reliably separate the two types of gaps.

Returning to k-means clustering, I adapted the method to analyse the distances between peaks rather than their absolute positions. By clustering these distances into "small" and "large" groups, I could determine the number of large distances present in the data, which provided a

direct estimate of the number of multiplets. The approach worked by counting large gaps as separators between clusters, allowing the algorithm to generate an initial guess for the number of clusters without manual input. Users could still adjust this automated estimate if needed.

Additionally, I extended the use of k-means to automate intensity (Y-axis) thresholding. By clustering intensity values, the algorithm identified a natural boundary between "noise" and "signal." This feature gave users a baseline threshold to modify as needed, enhancing the program's intuitiveness and significantly reducing the need for trial and error. Instead of starting from scratch, users could fine-tune an automatically generated threshold, streamlining the analysis process.

I also implemented an auto-scaling feature where users can enter their adjusted value in the command line, and the program automatically scales it according to the graph's Y-axis. If the scale is in the order of 10^7, for example, users can simply input 1, and the program will correctly apply the corresponding scaled value. This adjustment proved particularly useful, as I found manually entering multiple zeroes during testing tedious and prone to error.

I updated the plotting function to replot the graph after clustering, showing clusters in different colours. The same plotting algorithm was reused, but if cluster information is provided, it automatically assigns distinct colours to each cluster. This visualisation makes it easy for users to see if some peaks are incorrectly clustered and adjust the number of clusters accordingly, improving ease of use of the program.

4.5 Evaluating Clustering Algorithms

After validating the program's core functions with a simple, noise-free dataset, I tested it on more complex lab data I had personally collected from past organic chemistry labs. As the complexity of the datasets increased, I evaluated different clustering algorithms to determine which method offered the most accurate multiplet detection.

Initially, k-means clustering continued to perform well, maintaining accuracy in peak detection and clustering. However, during testing with more complex datasets, I found that Ward's hierarchical clustering provided slightly better performance. Ward's method demonstrated improved precision in grouping peaks, for example k-means might misclassify a quartet as a triplet and a singlet. Consequently, I replaced k-means with Ward's method for the primary clustering of multiplets. I also considered DBSCAN (Density-Based Spatial Clustering of Applications with Noise) due to its ability to detect clusters without needing a predefined number of clusters. However, DBSCAN required parameters like epsilon (the neighbourhood radius) and the minimum number of points per cluster, which varied significantly between datasets. This variability necessitated manual input, which conflicted with the goal of automating the program, so I excluded DBSCAN from the final implementation.

Despite Ward's improved accuracy in final multiplet clustering, it was not well-suited for initial cluster estimation. Ward's method is sensitive to outliers and does not provide a straight-

forward method for determining the initial number of clusters. Therefore, I continued using k-means clustering for initial cluster estimation and intensity (Y-axis) thresholding. K-means was particularly effective for these tasks, offering a quick and automated method for estimating the number of multiplets by analysing the distances between detected peaks. It also helped set a baseline intensity threshold by clustering intensity values into "noise" and "signal" groups.

The final program combined the strengths of both algorithms: k-means was used for initial cluster estimation and intensity thresholding, while Ward's hierarchical method handled the final clustering of multiplets. This hybrid approach allowed the program to leverage the automated and fast estimation capabilities of k-means with the superior clustering accuracy of Ward's method for non-spherical data distributions.

4.6 Calculating All Possible Multiplets

In literature, deep learning approaches have been proposed to automatically cluster and label NMR peaks with detailed multiplicities, including complex patterns like doublets of triplets or quartets of doublets.[5] These models can predict complex labels such as `"ddt"` (doublet of doublets of triplets) and more. However, these methods require extensive training and may not generalise well to all datasets. The current version of the code counts the number of peaks in a cluster and only generates basic multiplicity labels like `"d"` (doublet), `"t"` (triplet), and `"q"` (quartet). I wanted to avoid the risk of mislabelling complex patterns by generating all possible valid multiplicity labels for a given number of detected peaks. This way, a user can apply their own chemical knowledge by looking at the peaks and possible multiplicities and pick one, rather than receiving an incorrect value. Having the multiplicities possible would help them quickly determine which one is correct by looking at the shape of the peaks, rather than having to count to determine exactly which multiplicity it is and looking at its shape.

A straightforward approach to this problem would involve calculating all possible combinations of multiplicities that could sum to the total number of detected peaks. For example, for a peak count of 6, this method would generate every possible combination of factors (e.g., `"dd"` for a doublet of doublets, `"tq"` for a triplet of quartets, etc.). While this "brute force" method ensures no multiplicity is overlooked, its computational complexity is $\mathcal{O}(n^2)$, making it impractical for large peak counts due to exponential growth in possibilities. To improve efficiency, I developed a backtracking algorithm within the `factorize_multiplicity` function. Unlike the naïve approach, backtracking only explores valid paths, abandoning any combinations that cannot lead to a feasible multiplicity label early in the process. This approach significantly reduces the number of computations, improving both speed and resource usage, particularly for higher peak counts.

The algorithm applies specific rules to maintain accuracy and avoid generating misleading labels. If the number of detected peaks exceeds 36, the function defaults to `"multiplet"`, as this count is too high for reliable first-order multiplicity labelling. For smaller peak counts, the algorithm is more nuanced. When there are seven or fewer peaks, the function considers

all possible first-order multiplicities such as `"s"` (singlet), `"d"` (doublet), `"t"` (triplet), up to `"septet"`. For peak counts above seven, the algorithm restricts labels to the most common multiplicities in higher-order spectra, generating combinations using only `"d"`, `"t"`, and `"q"` (e.g., `"dd"`, `"dt"`, `"qt"`). This limitation is based on the likelihood that complex labels such as `"septet of triplets"` are more often due to incorrect clustering rather than a true spectral pattern. This approach improves the reliability of the generated labels, avoiding misclassification instead labelling it as a multiplet.

4.6.1 Backtracking Algorithm

Backtracking is a recursive algorithmic technique that incrementally builds a solution by exploring potential options one step at a time. If a certain path is determined to be invalid, it "backtracks" by undoing the last step and trying the next possibility. This method is particularly useful for problems that involve generating combinations, permutations, or subsets.

In the `factorize_multiplicity` function, the backtracking function is designed to generate all valid sequences of factors for a given `num_peaks`. It tries each allowed factor, verifies divisibility, and recursively continues with the reduced value. If a valid sequence is completed, it is added to the results. If not, the function backtracks using the `pop` method, removing the last added factor and exploring the next option.

Listing 2: Backtracking Code Snippet

```
def backtrack(current, path):
    if current == 1:
        results.append(path[:])
        return
    for f in factor_range:
        if current % f == 0:
            path.append(f)
            backtrack(current // f, path)
            path.pop()  # Backtrack by removing the last element
```

4.6.2 Example Walkthrough

For `num_peaks = 8`, the algorithm executes as follows:

- **Initial State:** `current = 8`, `path = []`
- **First Factor:** Factor 2 is valid (`8 % 2 == 0`), update to `current = 4`, `path = [2]`
- **Second Factor:** Factor 2 is valid again, `current = 2`, `path = [2, 2]`
- **Third Factor:** Factor 2 again, `current = 1`, `path = [2, 2, 2]`
- **Valid Sequence Found:** Add `[2, 2, 2]` to results, labels = `["ddd"]`

- **Backtrack:** `path.pop()` removes last 2, `path = [2, 2]`
- **Backtrack Again:** `path.pop()` removes 2, `path = [2]`
- **Backtrack Yet Again:** `path.pop()` removes 2, `path = []`
- **Try Next Factor:** Factor 4 is valid, `current = 2`, `path = [4]`
- **Next Factor:** Factor 2 is valid, `current = 1`, `path = [4, 2]`
- **Add Result:** Add `[4, 2]` to results, labels = `["ddd", "dq"]`
- **Backtrack:** Pop to `path = [4]`, then pop again to `path = []`
- **Final Path:** Try factor 2 first, then 4: `[2, 4]`, labels = `["ddd", "dq", "qd"]`

This approach offers a practical and systematic way to generate possible multiplicities, avoiding the risk of misidentification and ensuring that all valid options are considered. The method also provides a good balance between automation and user control, allowing researchers to validate the proposed multiplicities based on their domain knowledge.

4.7 Calculating J Coupling Values

Following the identification of peak clusters, the next step was to calculate J coupling values, which are essential in NMR research for interpreting spin-spin interactions. Traditionally, the expected number of J values correlates with the multiplicity of a peak—for instance, a triplet has a single J value, while a doublet of triplets (dt) would present two distinct J values. However, since the program does not pre-assign a specific multiplicity, all adjacent J values are calculated to avoid potential misclassification.

The function `calculate_j_values` was implemented to achieve this. It works by sorting the detected peaks, computing the differences between consecutive peaks (in parts per million, ppm), and converting these differences to Hertz (Hz) using the spectrometer frequency. By default, the frequency is set to 400 MHz, but users can adjust this parameter to match their experimental setup. This flexibility ensures accurate J values regardless of the spectrometer used.

By calculating all possible adjacent J values, the program provides a comprehensive dataset for researchers, allowing them to manually interpret them to determine the true multiplicity of each peak cluster.

4.8 Predicting J Value Intensities Using NMR Splitting Tree Diagrams

In NMR spectroscopy, splitting tree diagrams are commonly used to visualise how J couplings create specific splitting patterns in the spectrum for a given multiplicity. These diagrams help

illustrate the relative strengths of J values between adjacent peaks, providing valuable insights into the multiplicity of signals.

An example of a splitting tree for a triplet of doublets (`td`) is shown below in Figure 4. In this pattern, the stronger coupling (the triplet) creates larger groupings of peaks, while the weaker coupling (the doublet) introduces finer splitting within each triplet. This visual representation makes it easier to understand how each J coupling contributes to the observed pattern and which J values will appear stronger or weaker in the spectrum.

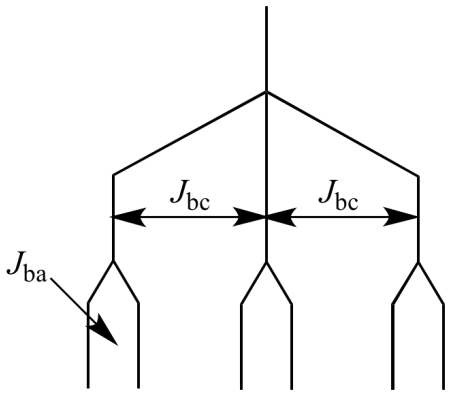


Figure 4: Example of an NMR splitting tree diagram for a triplet of doublets (`td`), showing the stronger triplet coupling (J_{bc}) and the weaker doublet coupling (J_{ba}). Figure from Hardinger[2]

.

It is possible to draw such splitting trees for many multiplicities to determine the pattern of J value strengths across adjacent peaks. By analysing these patterns, researchers can more accurately interpret the complex splitting observed in experimental NMR data.

To automate this process, I developed an algorithm that predicts the expected splitting pattern and relative J value intensities for any given multiplicity. The function `generate_j_pattern` uses a recursive approach within the `generate_splitting_path` method to construct a theoretical splitting path similar to those visualised in splitting tree diagrams. This approach provides a systematic way to generate all possible J value patterns, enhancing the robustness of NMR data analysis.

4.9 Generating Splitting Patterns for J Value Prediction

When developing the algorithm to predict J value splitting patterns, a method was needed to generate a splitting path that accurately reflected how multiplicity patterns work in NMR spectroscopy. The approach involved creating a recursive algorithm capable of handling any multiplicity pattern, from simple doublets to more complex cases like "dt" (doublet-triplet). The resulting function, `generate_j_pattern`, uses a recursive approach within the `generate_splitting_path` function to construct the splitting path.

4.9.1 Concept Behind the Splitting Path Algorithm

The splitting path algorithm was developed with the goal of mimicking the NMR splitting tree diagram, where inner couplings split first, and outer couplings further divide each sub-peak. Initially, I considered directly replicating the tree structure, calculating the separation of nodes at each level to match the J values' strength. However, this approach proved overly complex and difficult to generalise for higher-order multiplicities. Instead, I realised that the splitting pattern could be represented more intuitively using a matrix-like structure, where the second index controls the vertical position (simulating the deeper splitting) and the first index determines the horizontal placement (maintaining the main split pattern). This matrix approach not only simplified the algorithm but also allowed easy extension to more complex multiplicities. By leveraging this method, the algorithm efficiently generates all possible splitting paths while preserving the natural order and strength of J couplings observed in NMR spectra.

4.9.2 Recursive Algorithm: `generate_splitting_path`

The `generate_splitting_path` function uses a recursive method to generate all possible index combinations for a given set of splitting dimensions. It constructs the path by starting with the inner dimensions and progressively adding outer dimensions, alternating the direction of traversal to avoid diagonal transitions.

Listing 3: Function to generate expected rank order of J values based on a multiplicity pattern.

```
def generate_splitting_path(dimensions):
    if not dimensions:
        return
    if len(dimensions) == 1:
        for i in range(dimensions[0]):
            yield (i,)
    else:
        first_dim = dimensions[0]
        remaining_dims = dimensions[1:]
        subpath = list(generate_splitting_path(remaining_dims))
        reversed_subpath = subpath[::-1]
        for i in range(first_dim):
            if i % 2 == 0:
                for combo in subpath:
                    yield (i,) + combo
            else:
                for combo in reversed_subpath:
                    yield (i,) + combo
```

How the Algorithm Works The algorithm works as follows:

- **Base Case**: If the dimensions list is empty, the function terminates immediately. If only one dimension is present, it generates sequential indices as single-element tuples.

- **Recursive Case**: The function decomposes the dimensions into the first dimension (`first_dim`) and the remaining dimensions (`remaining_dims`).
  - It generates a subpath recursively for the remaining dimensions.
  - Creates both the normal and reversed versions of this subpath.
  - Iterates through indices of the first dimension:
    * For even indices, it appends the normal subpath.
    * For odd indices, it appends the reversed subpath.

Example: Splitting Path for `dt` (Doublet-Triplet) For a multiplicity of `dt`, the dimensions are $[2, 3]$, representing a doublet (2 lines) followed by a triplet (3 lines). The function call:

```
list(generate_splitting_path([2, 3]))
```

Produces the path:

$$\begin{bmatrix} (0,0) & (0,1) & (0,2) \\ (1,2) & (1,1) & (1,0) \end{bmatrix}$$

Visualising the Splitting Path The splitting pattern can be visualised in a grid as follows:

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ 0 & \text{x} & \longrightarrow \text{x} & \longrightarrow \text{x} \\ & & & \downarrow \\ 1 & \text{x} & \longleftarrow \text{x} & \longleftarrow \text{x} \end{array}$$

- **First row** ($i = 0$): The function uses the normal subpath, resulting in $(0, 0)$, $(0, 1)$, $(0, 2)$.
- **Second row** ($i = 1$): The function uses the reversed subpath, giving $(1, 2)$, $(1, 1)$, $(1, 0)$.

4.9.3 Converting the Splitting Path to J Value Ranks

The `generate_j_pattern` function uses the generated splitting path to determine the rank order of J values. It compares each pair of consecutive indices to identify which dimension changed, assigning ranks accordingly.

Rank Assignment Method The function evaluates each step in the path and determines which dimension is responsible for the change. The dimension that changes dictates the rank assigned to that step, with higher dimensions receiving higher ranks, corresponding to stronger J couplings.

Example of Rank Assignment For the splitting path generated for `dt`:

$$\begin{bmatrix} (0,0) & (0,1) & (0,2) \\ (1,2) & (1,1) & (1,0) \end{bmatrix}$$

The function compares each step:

- $(0,0)$ to $(0,1)$: Second dimension changes $\rightarrow$ Rank 1
- $(0,1)$ to $(0,2)$: Second dimension changes $\rightarrow$ Rank 1
- $(0,2)$ to $(1,2)$: First dimension changes $\rightarrow$ Rank 2
- $(1,2)$ to $(1,1)$: Second dimension changes $\rightarrow$ Rank 1
- $(1,1)$ to $(1,0)$: Second dimension changes $\rightarrow$ Rank 1

The computed rank order of J values is:

$$[1, 1, 2, 1, 1]$$

This rank order reflects the expected pattern of coupling constants, allowing for precise matching of observed J values with theoretical multiplicity patterns.

4.10 Matching J Values to Multiplicity Patterns

The function `match_j_values_to_multiplicity` compares observed J values with the predicted splitting patterns of all possible multiplicities. It first uses DBSCAN (Density-Based Spatial Clustering of Applications with Noise) to cluster the observed J values. DBSCAN is particularly useful as it does not require a predefined number of clusters and can effectively handle outliers as noise. An important feature of this function is the adjustable uncertainty parameter, which controls how closely J values must align to be grouped into the same cluster. For example, with an uncertainty setting of 1, J values of [7.1, 7.1, 9] would be clustered into two groups: [7.1, 7.1] and [9]. These clusters are then ranked by strength, with the most common cluster assigned the lowest rank. The resulting rank order would be [1, 1, 2], indicating that the first two J values are similar, while the third represents a distinctly stronger coupling.

The function then directly compares the ranked observed J values to the predicted patterns generated by the `generate_j_pattern` function for each possible multiplicity. Since each valid multiplicity has a unique splitting pattern, the algorithm checks for an exact match between the observed ranks and the predicted pattern. If a match is found, the function assigns the corresponding multiplicity label along with the averaged J values for the matching clusters. However, if no predicted pattern matches the observed ranks, the function returns `'multiplet'`. This approach ensures that the function only assigns a specific multiplicity when there is clear evidence,

reducing the risk of misclassification, especially in complex or noisy datasets. By allowing users to adjust the clustering sensitivity, the method provides flexibility to accommodate different data qualities and experimental conditions.

4.11 Functional Group Annotation

An additional feature was included to annotate the clusters with possible functional groups based on their ppm ranges. For example, aromatic groups typically appear around 7–8 ppm, while aldehydic groups are in the 9–10 ppm range. This annotation provides insights into potential structural contexts directly from the spectral data, reducing the need for manual cross-referencing with chemical shift tables. A dictionary from `https://www.ucl.ac.uk/nmr/sites/nmr/files/L2_3_web.pdf` was used for the reference data.

4.12 Output Files and User Interaction

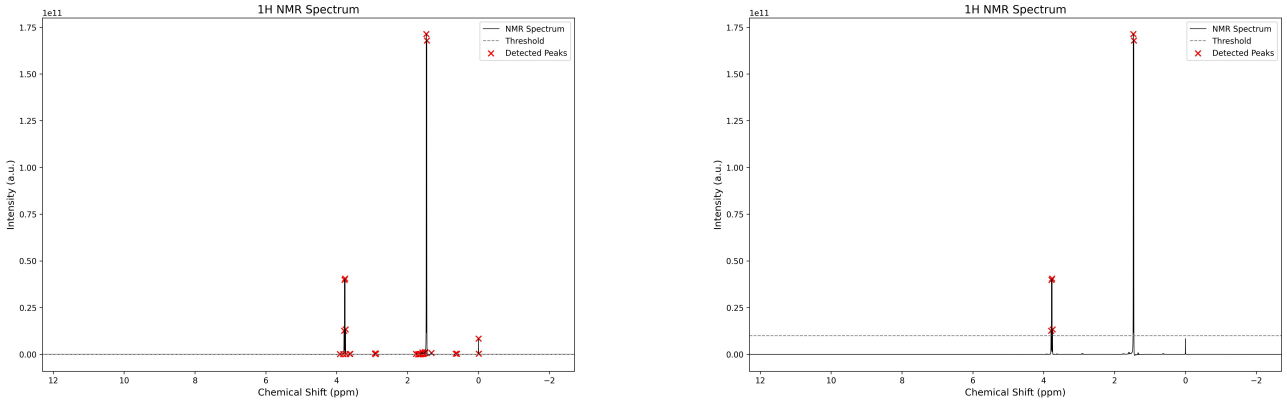
The program outputs analysis results into two files, both named using the input filename and saved in the `nmrdata_output/` directory. These files are: a detailed file (e.g., `[filename]_detailed_calculation.txt`) and a simple file (e.g., `[filename]_simple_calculation.txt`). The detailed file provides comprehensive information about each cluster, including precise peak positions, all J values, and assigned functional groups. This is particularly useful for users who want to manually inspect the J values and verify the analysis process step-by-step. The simple file offers a summarised view, showing only the peak range or average peak value, identified multiplicity, averaged J values, and relevant functional groups, which is ideal for quick reviews or sharing high-level results with collaborators. If the files already exist, they are automatically overwritten upon rerunning the program. Additionally, the program saves the `matplotlib` plot as an image file using the same filename convention, enabling users to easily review and share the analysis results without needing to rerun the program.

The program allows users to iteratively adjust the analysis parameters, specifically the threshold for peak picking and the number of clusters (k). After reviewing the initial results, users can modify these inputs and re-run the analysis until they are satisfied with the output. The program replots the data, helping users understand how changes to the threshold or cluster count affect the analysis. During testing, I found that k-means clustering often underestimates the number of multiplets when some are closely spaced while others are far apart. To address this, I introduced an "extra" parameter (defaulting to 10) that allows the program to test a range of cluster counts and select the best fit based on minimising ambiguous "multiplet" classifications.

The "extra" parameter is only applied when the program automatically estimates the number of clusters on the initial run. It adds the specified number to the initial estimate and tests all possible cluster counts within this range. For example, if the estimated cluster count is 5 and "extra" is set to 10, the program will evaluate cluster counts from 5 to 15. This feature

is particularly useful for complex datasets where a more extensive search may be needed to accurately determine the correct number of multiplets. When a user specifies a cluster count manually, the "extra" parameter is ignored, and the program strictly adheres to the user's input. This design ensures that users with higher computational power can choose a larger "extra" value, allowing the program to explore a broader range of possible cluster counts and potentially achieve more accurate results.

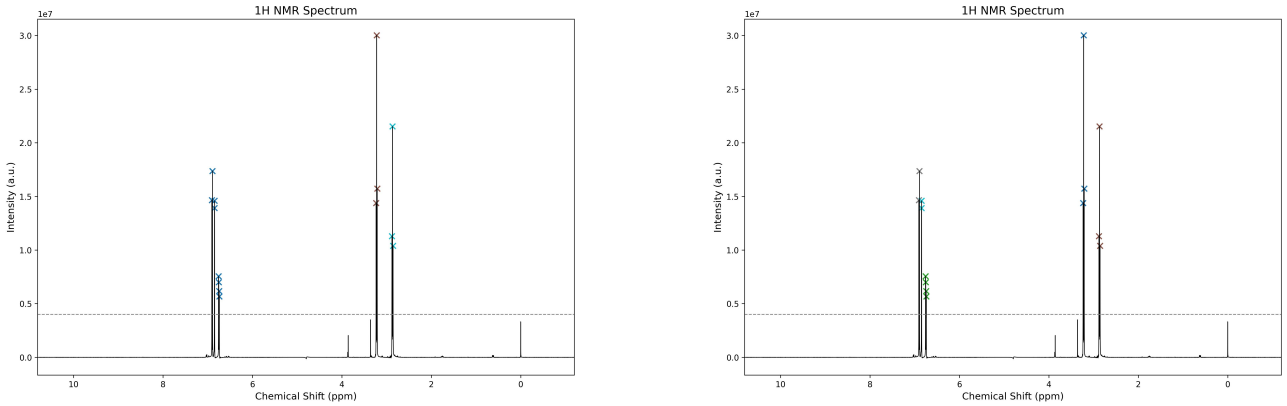
Figure 5 shows how applying a threshold of 0.1 improves peak detection by removing background noise (example from Test 2^6).



(a) Before Thresholding **(b)** After Thresholding

Figure 5: Comparison of image before and after applying thresholding to 0.1. Clearer peak identification is achieved post-thresholding.

Figure 6 shows the effect of increasing the number of clusters from 3 to 5 (example from Test 4^7). The clusters are highlighted in different colours.



(a) Using 3 Clusters **(b)** Using 5 Clusters

Figure 6: Effect of increasing the number of clusters from 3 to 5. Peaks are better clustered, allowing the multiplicity to be calculated

I considered adding a tolerance parameter that would allow small discrepancies in peak counts (e.g., treating 11 peaks as 12 for pattern matching). However, this approach risked introducing incorrect multiplicity assignments, as it essentially allowed mismatches to fit expected patterns.

Instead, I concluded that this issue would be better addressed by advanced smoothing algorithms in future work. Such methods could more effectively manage noise and peak broadening, allowing the program to maintain accuracy without relying on potentially misleading tolerance settings.

The program also prints simplified data with just the average peaks, multiplicity and j values to terminal. This ensures that users can verify correct functionality without opening external files. The terminal output provides immediate feedback on detected peaks and multiplicities, making the analysis process to reselect the number of clusters more efficient.

4.13 Development of the J-Value Calculation Algorithm

During testing, several cases showed that the algorithm struggled to correctly identify and assign J-values for doublet of doublets (dd) patterns. In particular, the following tests demonstrated failures in distinguishing dd multiplicities:

Test	Expected Output (J values in Hz)	Actual Output
Test 3[8]	0.886 (t, J = 7.47 Hz), 1.641 (m), 1.734 (m), 3.990 (dd, J = 6.56, 4.52 Hz)	0.00 (s), 0.87 (t, J = 7.45 Hz), 1.62 (hex, J = 7.09 Hz), 1.71 (m), 3.98 (m)
Test 6[9]	1.99 (m), 2.06 (m), 2.34 (m), 3.33 (dt, J = 14.02, 7.11 Hz), 3.41 (dt, J = 11.65, 7.02 Hz), 4.12 (dd, J = 8.63, 6.42 Hz)	0.00 (s), 1.96 (m), 2.32 (m), 3.30 (m), 3.39 (m), 4.11 (m)

Table 1: Table showing selected tests for complex multiplicity calculations. Table by the author.

In **Test 3[8]**, the program failed to classify a dd multiplicity, instead outputting it as a multiplet. Debugging revealed that the J-values were too close together, leading to an inversion in the expected intensity order. Similarly, in **Test 6[9]**, the algorithm was unable to distinguish between a dd and a dt pattern, defaulting to "multiplet" due to improper spacing of J-values. The issue stemmed from relying on adjacent peak differences rather than analyzing splitting hierarchies.

4.13.1 Development of `calculate_j_vals`

To address these limitations, an approach was developed to ensure correct J-value assignment within a given multiplicity pattern. Instead of computing purely adjacent peak differences, the algorithm was redesigned to systematically divide peaks based on the multiplicity structure.

This ensures that J-values are extracted from relevant peak groupings rather than as simple nearest-neighbor differences.

The function `calculate_j_vals` follows a structured method to extract J-values from a given set of NMR peaks. The algorithm begins by sorting the peaks to ensure correct difference calculations. It then processes the multiplicity from right to left, meaning that in a pattern like 'td', the function first handles the doublet splitting, then the triplet. This reversed order allows the function to correctly reconstruct the hierarchical splitting of a multiplet. Each iteration divides the peak list into subgroups of equal size based on the current multiplicity component. Within each subgroup, J-values are computed as differences between adjacent peaks and converted into Hz using the provided spectrometer frequency. These values are placed into a structured list according to a ranking pattern, ensuring that each splitting rank is correctly accounted for. After each iteration, the peak list is reduced by replacing each subgroup with its mean position, allowing the next iteration to operate on fewer peaks.

The function uses `generate_j_pattern` to determine the correct placement of J-values within the final output. This is necessary because, unlike simple adjacent difference calculations, J-values in a complex multiplet structure follow a hierarchical order. By following a predefined pattern, the function ensures that J-values from different ranks of splitting are placed correctly. This maintains consistency with expected NMR multiplicity patterns and prevents misalignment of J-values in cases where multiple couplings exist.

Listing 4: Function to calculate NMR J-values based on peak patterns and multiplicity.

```
def calculate_j_vals(peaks, multiplicity, frequency):
    line_counts = multiplicity_to_line_count(multiplicity)

    if len(line_counts) == 1:
        return _adjacent_j_values(peaks, frequency)

    line_counts_reversed = line_counts[::-1]
    current_peaks = peaks.copy()
    rank_pattern = generate_j_pattern(multiplicity)
    final_j_vals = [0] * len(rank_pattern)

    for iteration, n in enumerate(line_counts_reversed):
        rank_number = iteration + 1
        n_groups = len(current_peaks) // n
        groups = [current_peaks[i * n:(i + 1) * n] for i in range(n_groups)]

        # Compute J-values (Hz) from peak differences within each subgroup
        j_vals_iteration = [(np.diff(g) * frequency).tolist() for g in
        groups]
        j_vals_iteration = [j for sublist in j_vals_iteration for j in
        sublist]

        # Assign J-values to correct positions based on rank pattern
        indices_to_fill = [i for i, rank in enumerate(rank_pattern) if rank
```

```
            == rank_number]
            for idx, val in zip(indices_to_fill, j_vals_iteration):
                final_j_vals[idx] = val

            # Replace each subgroup with its mean to form new representative
            peaks
            current_peaks = np.array([np.mean(g) for g in groups])

    return final_j_vals
```

4.13.2 Doublet of Doublets Example

To demonstrate how `calculate_j_vals` works, consider a simple case of a *doublet of doublets* (multiplicity = `dd`), which gives rise to four peaks. Suppose we are given the following NMR peak positions:

- `peaks=[1.00,1.01,1.04,1.05]` (in ppm)
- `multiplicity="dd"`
- `frequency=400.0` (MHz)

The function interprets this as a hierarchical splitting into 2×2 peaks. Internally, it reverses the multiplicity string to process the rightmost splitting first, in this case the second doublet.

First Iteration (Rightmost 'd'):

- Input peaks: `[1.00,1.01,1.04,1.05]`
- Grouped into subgroups of 2: `[[1.00,1.01],[1.04,1.05]]`
- Compute J-values (Hz): `(1.01-1.00)*400=4.0`, `(1.05-1.04)*400=4.0`
- Resulting J-values: `[4.0,4.0]`
- Replace each group with its mean to form new peaks: `[1.005,1.045]`

Second Iteration (Leftmost 'd'):

- Input peaks: `[1.005,1.045]`
- Grouped into one subgroup: `[[1.005,1.045]]`
- Compute J-value: `(1.045-1.005)*400=16.0`
- Resulting J-values: `[16.0]`

The function then uses `generate_j_pattern("dd")` to determine the correct placement of J-values by splitting rank. For this case, the pattern might be `[1,1,2]`, which assigns the two 4.0 Hz couplings to rank 1 and the 16.0 Hz coupling to rank 2. The final output is: `[4.0,4.0,16.0]`

This structured approach ensures correct identification of couplings even when peaks are not evenly spaced, as it avoids naive adjacent subtraction. By grouping peaks and using the mean positions for further iterations, this method better handles hierarchical splitting. This approach is therefore more robust than calculating differences between adjacent peaks, and is why it now succeeds in previously failing test cases such as Test 3.

An alternative approach was considered, where J-values would be dynamically assigned based on subgroup structures rather than relying on a predefined rank pattern. This method would have involved determining splitting hierarchies based on peak distributions, where groups would first be identified, and then J-values would be calculated between them. For example, in a 'td' pattern, the triplet would be identified first, dividing the peaks into three primary subgroups, each containing a doublet. J-values would then be extracted first between the main triplet subgroups before measuring intra-group splittings. However, this approach proved too complex and did not provide a significant advantage over the previously implemented method.

The `match_j_values_to_multiplicity` function was amended to ensure that if the J values match multiple multiplicities, the program outputs 'multiplet' rather than selecting one. This reduces the risk of misclassification by acknowledging ambiguity in cases where the data does not clearly indicate a single multiplicity.

4.14 Final Flowchart

The final program flowchart (Figure 7) shows the updated structure of the program following iterative development and refinement. In contrast to the initial version, this flowchart includes significantly more logic related to multiplicity classification and J-value analysis. Additions include the generation of all valid multiplicity patterns using backtracking, the calculation and ranking of J-values based on simulated NMR splitting trees, and the use of pattern-matching algorithms to compare observed data with theoretical expectations. These changes reflect the shift in focus from simple peak detection to a more sophisticated analysis that supports accurate and automated interpretation of complex NMR spectra.

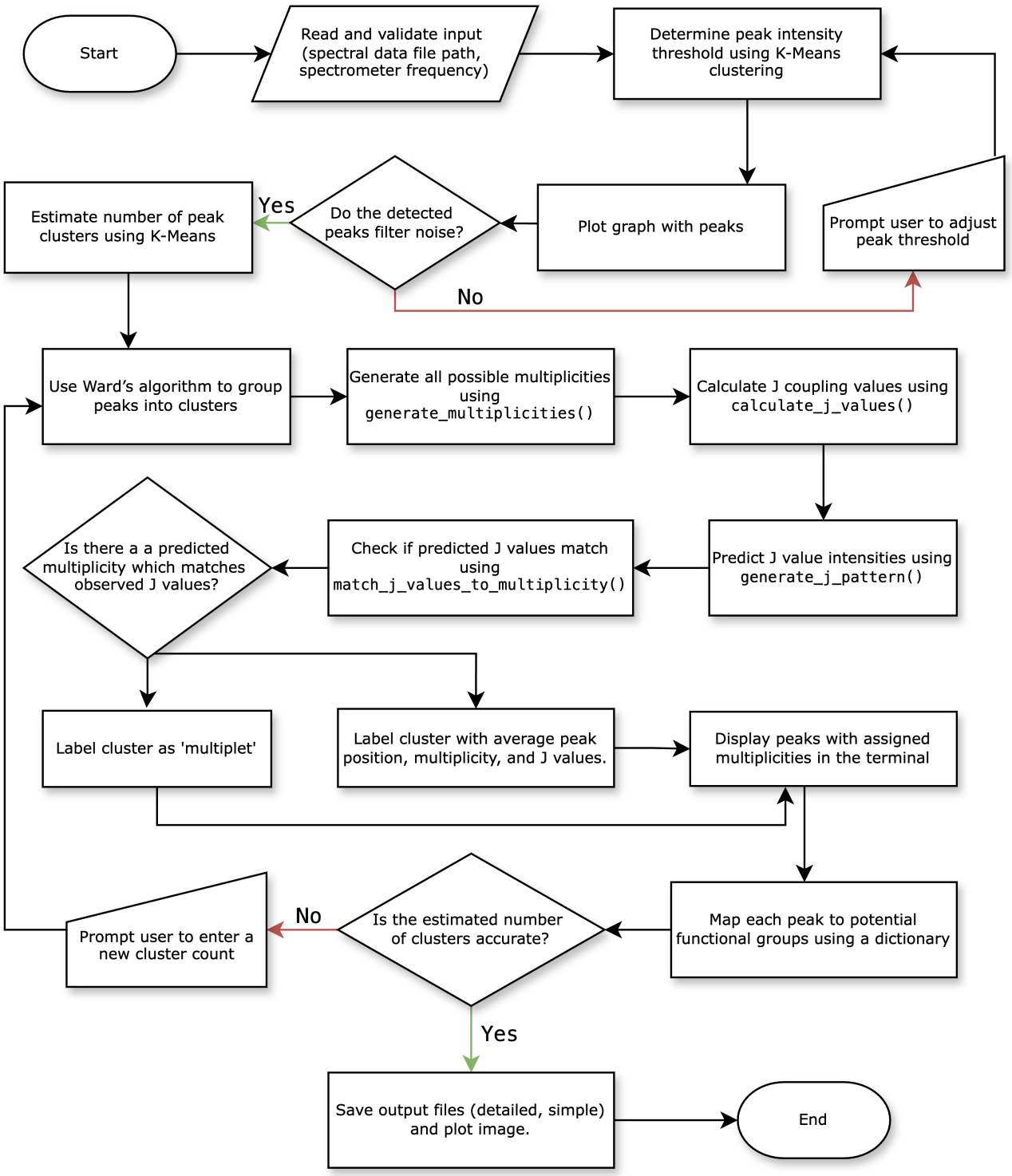


Figure 7: Final program flowchart showing the complete structure of the NMR analysis pipeline, including peak detection, clustering, J-value calculation, and multiplicity classification. Figure by the author.

5 Tests

Test	Expected Output (J values in Hz)	New Output Results
Test 1[10]	8.12 (s, J = 2 Hz), 8.11 (s, J = 2 Hz)	3.02 (s), 3.92 (s)
Test 2[6]	1.47 (d, J = 7.28 Hz), 3.77 (q, J = 7.23 Hz)	1.46 (d, J = 7.22 Hz), 3.75 (m)
Test 3[8]	0.886 (t, J = 7.47 Hz), 1.641 (m), 1.734 (m), 3.990 (dd, J = 6.56, 4.52 Hz)	0.00 (s), 0.87 (t, J = 7.45 Hz), 1.63 (m), 1.71 (td, J = 4.56, 7.20 Hz), 3.98 (dd, J = 4.52, 6.62 Hz)
Test 4[7]	2.87 (t, J = 7.24 Hz), 3.22 (t, J = 7.24 Hz), 6.75 (dd, J = 8.07, 2.11 Hz), 6.85 (d, J = 2.11 Hz), 6.90 (d, J = 8.07 Hz)	2.85 (t, J = 7.25 Hz), 3.21 (t, J = 7.25 Hz), 6.74 (dd, J = 2.09, 8.13 Hz), 6.84 (d, J = 2.09 Hz), 6.89 (d, J = 8.13 Hz)
Test 5[11]	3.81 (dd, J = 12.79, 4.51 Hz), 3.92 (m), 4.11 (m), 4.20 (t, J = 5.67 Hz), 4.30 (t, J = 4.67 Hz), 5.89 (d, J = 3.98 Hz), 6.05 (d, J = 7.55 Hz), 7.83 (d, J = 7.60 Hz),	0.00 (s), 3.79 (dd, J = 4.49, 12.82 Hz), 3.90 (m), 4.10 (hept, J = 2.23 Hz), 4.18 (t, J = 5.68 Hz), 4.29 (t, J = 4.67 Hz), 5.89 (d, J = 4.03 Hz), 6.04 (d, J = 7.60 Hz), 7.82 (d, J = 7.60 Hz)
Test 6[9]	1.99 (m), 2.06 (m), 2.34 (m), 3.33 (dt, J = 14.02, 7.11 Hz), 3.41 (dt, J = 11.65, 7.02 Hz), 4.12 (dd, J = 8.63, 6.42 Hz)	0.00 (s), 1.96 (m), 2.32 (m), 3.30 (m), 4.11 (dd, J = 6.51, 8.71 Hz)

Table 2: Performance of the NMR analysis program across multiple test datasets, highlighting accuracy in peak detection, clustering, and multiplicity classification. Table by the author.

The spectral data used for testing and validation was sourced from the Human Metabolome Database (HMDB) at `https://hmdb.ca/metabolites`. The HMDB is a comprehensive, peer-reviewed repository containing experimentally acquired ^{1}H NMR spectra for thousands of human metabolites. All test cases selected for this project were based on experimental spectra of known metabolites, ensuring that the program was evaluated against realistic and biologically

relevant data.

Test 1[10] tests the detection of two singlets in a simple dataset, ensuring the program correctly identifies peaks without complex splitting. The threshold was adjusted to 1 to improve peak detection. The program successfully identified both peaks and avoided over-clustering, producing results that matched the HMDB database.

Test 2[6] tests whether the program can correctly identify simple multiplets, specifically distinguishing between a doublet and a quartet. The threshold was set to 0.1 manually as auto-picking resulted in values that were too low. The program misclassified the quartet as a multiplet because it matched both a doublet of doublets (dd) and a quartet (q). This happened because the new calculate J values algorithm, which calculates J values based on multiplicity, treated the quartet as two adjacent dd rather than a single unit. In principle, this should not have happened because a quartet has equal spacing between all peaks, meaning the three calculated J values should remain the same whether computed between the mean of two doublets or across the whole quartet. However, the classification still failed, highlighting a limitation of the new approach.

Test 3[8] increases complexity by introducing both dd and non-split multiplicities, testing whether the program can correctly identify defined multiplicities while handling peaks that do not follow simple patterns. The threshold was set to 0.1, with five clusters specified. In Table 1, the previous version struggled with dd classification, often mislabeling them as multiplets. The new calculate J values algorithm improved handling of closely spaced J values, successfully classifying dd and demonstrating an improvement over Test 2. The 0.00 (s) label is a reference line which my program picked up and was skipped out from the HMDB labels.

Test 4[7] evaluates whether the program can correctly identify triplets and dd when peaks are closely spaced. The threshold was set to 0.4, and five clusters were specified. The program correctly identified all peaks, including dd, confirming its ability to distinguish between different types of complex multiplicities when peak separation is clearer.

Test 5[11] involves a dataset with increased complexity, testing whether the program can handle overlapping J values and peak separation when clusters are closely spaced. The auto-selected number of clusters was eight. After increasing it to nine and adjusting uncertainty to two, the triplet was correctly identified, which was not classified correctly at uncertainty 1. Other peaks remained unchanged, showing that uncertainty tuning helped resolve specific cases rather than broadly improving classification.

Test 6[9] is the most complex dataset, filled with only complicated multiplicities, testing whether the program can correctly distinguish between dd and multiplets when no simple splitting patterns are present. The program correctly identified the dd, but all remaining peaks were classified as multiplets. Compared to Table 1, this is an improvement, as previous versions failed to recognize dd at all, showing that the new calculate J values algorithm has helped in some cases. However, it still struggled to fully differentiate the remaining peaks, suggesting that further refinements are needed for more complicated splitting patterns.

The uncertainty parameter was kept at its default value of 1, except in Test 5, where increasing it to 2 improved classification of the triplet. The extra parameter for cluster specification could have been used, but I preferred to manually set the number of clusters for better accuracy, though auto-clustering was still kept as an option. Since calculating J values based on multiplicity improved results in some cases but caused misclassifications in others, it may be useful to allow users to choose whether or not to apply this method in future versions, depending on their dataset.

These tests cover a range of cases, ensuring the program can handle singlets, simple multiplets, complex multiplicities, datasets with both, and cases where only complex multiplicities exist. They test how well the program differentiates J values, recognises splitting patterns, and adapts to changes in threshold and uncertainty. The HMDB database is composed of real experimental data, meaning the tests use actual lab data rather than simplified examples, ensuring that future versions of the program are evaluated against realistic conditions. This provides a reliable benchmark for verifying whether new functionality correctly handles singlets, multiplets, and a variety of splitting patterns, ensuring the program remains robust under different conditions.

The tests are stored in the `tests/` directory, with filenames indicating the number and the NMR frequency. For example, `hmdb_sample_1_f500.tsv` refers to test 1 taken from HMDB data, recorded at 500 MHz. Each filename follows the pattern `source_sampleNumber_fXXX.tsv`, where `XXX` denotes the spectrometer frequency in MHz.

6 Documentation

6.1 Installation

The program requires `Python 3.12.8` and the following external libraries:

`numpy==2.2.2`, `matplotlib==3.10.0`, `scikit-learn==1.6.1`

These can be installed using the following command:

```
pip install numpy==2.2.2 matplotlib==3.10.0 scikit-learn==1.6.1
```

Once installed, the program can be run from the command line. Ensure that the script is placed in the directory from which it will be executed.

6.2 Usage

The program is a command-line tool designed for NMR peak analysis. It processes a CSV file containing spectral data and determines peak multiplicities based on clustering and J-value analysis.

To run the program, use the following command:

```
python main.py --file [input_file] --frequency [MHz] [other_arguments]
```

Replace `[input_file]` with the path to the CSV file containing the NMR data, and `[MHz]` with the operating frequency of the spectrometer.

6.3 Command-line Arguments

The program accepts the following arguments:

- `--file`: Path to the input CSV file containing the NMR spectral data. The file should include columns for chemical shift (ppm) and intensity.
- `--frequency`: Operating frequency of the spectrometer in MHz. Used to convert peak separations from ppm to Hz for J-value calculations.
- `--extra`: Number of additional clusters to test beyond the automatically estimated number. This improves accuracy when both closely and widely spaced clusters are present.
- `--uncertainty`: Tolerance for matching calculated J-values. A larger value increases flexibility in multiplicity classification; a smaller value enforces stricter matching.

6.4 Example Usage

For an NMR dataset stored in `tests/hmdb_sample_4_f600.tsv`, recorded at 600 MHz:

```
python src/main.py --file tests/hmdb_sample_4_f600.tsv --frequency 600
```

To allow the program to test 10 additional clusters and reduce the uncertainty tolerance to 0.1:

```
python src/main.py --file tests/hmdb_sample_4_f600.tsv --frequency 600
--uncertainty 0.1 --extra 10
```

Note: The test data in the `tests/` directory follows the naming convention `fXXX`, where `XXX` denotes the recording frequency in MHz (e.g., `f600` = 600 MHz). The initial execution of the programme may require a few seconds to complete.

6.5 Output

The program produces four outputs:

1. `simple_calculations.txt`: Summarised detected peaks with their predicted multiplicities, J-values and possible functional groups.
2. `detailed_calculations.txt`: Detailed output including all peak positions, all J-values, and clustering information.

3. A graphical output showing the spectrum with detected peaks and multiplicities.

4. Terminal output for quick visual inspection.

All output files are saved in the `nmrdata_output/` directory. If the directory or files already exist, they are automatically overwritten. The outputs are named based on the original input filename.

6.6 Example Output

Test 4 Terminal Output:

^{1}H NMR (600 MHz)
δ 2.85 ppm (t, J = 7.25 Hz)
δ 3.21 ppm (t, J = 7.25 Hz)
δ 6.74 ppm (dd, J = 2.09, 8.13 Hz)
δ 6.84 ppm (d, J = 2.09 Hz)
δ 6.89 ppm (d, J = 8.13 Hz)

Generated Files:

1. `example_output/hmdb_sample_4_f600_nmr_plot.jpg`

2. `example_output/hmdb_sample_4_f600_simple_calculation.txt`

3. `example_output/hmdb_sample_4_f600_detailed_calculation.txt`

Note: All generated example files are stored in the example_output folder.

7 Self-reflection

I began with a strong foundation in programming and aimed to build a tool that could automate NMR peak analysis. I found implementing the peak-picking algorithm straightforward, and decided to extend the functionality by experimenting with clustering techniques. My first implementation was a version of k-means, which performed well on simple datasets. I then investigated additional algorithms, including Ward's method and DBSCAN. I was excited to learn about different clustering algorithms.

The next challenge was handling multiplicities. At first, I simply counted the number of peaks in a cluster and assigned a multiplicity based on that. For example, four peaks meant a quartet. This approach clearly failed in more complex cases like doublet of doublets (dd), where multiple splitting interactions produce the same number of peaks. I wanted a better solution and decided it was more useful to generate all valid multiplicities for a given peak count. This would allow users to apply their own chemical knowledge, rather than risk getting an incorrect prediction.

I initially wrote a brute-force method to list all combinations, but quickly realised this was inefficient. I remembered backtracking from STAT0041, a module on data structures and algorithms, and rewrote the function using a recursive backtracking approach. This reduced unnecessary computation and made it feasible to explore complex patterns more effectively. It was rewarding to take what I had learned in that module and use it directly in this project.

After that, I started thinking about whether the program could guess specific multiplicities like dd or dt by using the spacing between adjacent J-values. I realised that NMR splitting trees contain the information needed to make this possible. I wanted to mimic that logic in code. At first, I tried building an actual tree structure, but I found it difficult to manage. I thought I could use BFS/DFS but wasn't sure how to traverse it properly, and it became too complex to work with.

Eventually, I realised I could simulate the splitting behaviour using a matrix instead. This worked much better. By representing the splitting pattern as a series of row operations and using recursive logic, I could reconstruct the splitting path and calculate the J-values from it. This approach supported patterns like ddt and dddd and scaled well. I also learned about Python features like `yield`, which helped reduce memory usage during recursion. I couldn't find many other algorithmic solutions to this problem, most approaches I came across were deep learning based, so it was interesting to build something from scratch.

I used Git for version control throughout the project. This was particularly helpful when testing new features, as small changes in one part of the code could break other components. Version control allowed me to experiment without losing progress.

I think this project has the potential to become an open-source package. Many of the functions I wrote are not commonly available and could be useful to undergraduate chemistry students, especially when checking the correctness of hand-drawn splitting trees or interpreting new data. In the future, I would like to add native support for Bruker and JCAMP data formats, so that users can load raw spectral files directly without conversion.

One limitation I found was in automatic Y-axis thresholding. Often the threshold was too low, and I noticed that noise was more complicated than a single cutoff could handle. In the future, I would like to implement a more advanced filter, as described in sources like Liu et al.[4]

While the program is not perfectly accurate, I believe it is a meaningful step toward improving automated analysis of NMR spectra. It already helps speed up the labelling process, and with further development, the outputs could contribute to creating better training data for deep learning models.

AI Statement

No generative AI tools were used in the creation of this project.

Code Listing

cluster.py

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering

# Define empirical 1H chemical shift ranges
# https://www.ucl.ac.uk/nmr/sites/nmr/files/L2_3_web.pdf
CHEMICAL_SHIFT_RANGES = {
    "Aldehyde": (9.5, 10.5),
    "Aromatic": (6.5, 8.2),
    "Alkene": (4.5, 6.1),
    "Alkyne": (2.0, 3.2),
    "Acetal": (4.5, 6.0),
    "Alkoxy": (3.4, 4.8),
    "N-Methyl": (3.0, 3.5),
    "Methoxy": (3.3, 3.8),
    "Methyl": (0.9, 1.0),
    "Methyl (double bond/aromatic)": (1.8, 2.5),
    "Methyl (CO-CH3)": (1.8, 2.7),
    "Methylene (CH2-O)": (3.6, 4.7),
    "Methylene (CH2-R1R2)": (1.3, 1.4),
    "Methine": (1.5, 1.6),
    "Cyclopropane": (0.22, 0.25),
    "TMS (Reference)": (0.0, 0.0),
    "Metal Hydride": (-5, -20)
}

# Adapted from https://stanford.edu/~cpiech/cs221/handouts/kmeans.html
# Tolerance/ max_iter from https://scikit-learn.org/stable/modules/generated
/sklearn.cluster.KMeans.html
def kmeans_clustering(data, k, max_iter=300, tol=1e-4, random_state=42):
    """
    Perform K-Means clustering on the given dataset.

    Parameters
    ----------
    data : numpy.ndarray
        Data points to be clustered, shape (n_samples, n_features).
    k : int
        Number of clusters to form.
    max_iter : int, optional
        Maximum number of iterations of the k-means algorithm, by default
        300.
    tol : float, optional
        Tolerance to declare convergence, by default 1e-4.
```

```
    random_state : int, optional
        Random seed for reproducibility, by default 42.

    Returns
    -------
    labels : numpy.ndarray
        Cluster labels for each data point.
    centroids : numpy.ndarray
        Coordinates of cluster centroids.
    """
    np.random.seed(random_state)
    centroids = data[np.random.choice(data.shape[0], k, replace=False)].
    astype(float)
    for _ in range(max_iter):
        distances = np.linalg.norm(data[:, None] - centroids, axis=2)
        labels = distances.argmin(axis=1)
        old_centroids = centroids.copy()
        for i in range(k):
            pts = data[labels == i]
            centroids[i] = pts.mean(axis=0) if len(pts) else data[np.random.
            randint(0, data.shape[0])]
        if np.linalg.norm(centroids - old_centroids) < tol:
            break
    return labels, centroids

def hierarchical_clustering(peaks, n_clusters, linkage='ward'):
    """
    Perform hierarchical (agglomerative) clustering on NMR peaks.

    Parameters
    ----------
    peaks : numpy.ndarray
        Array of peak positions (1D).
    n_clusters : int
        Number of clusters to find.
    linkage : str, optional
        Linkage criterion ('ward', 'complete', 'average', 'single'), by
        default 'ward'.

    Returns
    -------
    labels : numpy.ndarray
        Cluster labels for each peak.
    centroids : numpy.ndarray
        Mean position of each cluster.
    """
    data = peaks.reshape(-1, 1)
    hc = AgglomerativeClustering(n_clusters=n_clusters, linkage=linkage)
    labels = hc.fit_predict(data)
```

```
    cluster_dict = {i: [] for i in range(n_clusters)}
    for peak, lbl in zip(peaks, labels):
        cluster_dict[lbl].append(peak)
    centroids = np.array([np.mean(val) for val in cluster_dict.values()])
    return labels, centroids


def load_csv_coordinates(file_path):
    """
    Load x and y coordinates from a CSV file.

    Parameters
    ----------
    file_path : str
        Path to the CSV file.

    Returns
    -------
    x : numpy.ndarray
        X coordinates.
    y : numpy.ndarray
        Y coordinates.
    """
    x, y = [], []
    with open(file_path) as file:
        next(file)  # Skip header
        for line in file:
            parts = line.split()
            x.append(float(parts[0]))
            y.append(float(parts[1]))
    return np.array(x), np.array(y)

def determine_noise_threshold(y_values, n_clusters=2):
    """
    Calculate a threshold to differentiate noise from signals using K-Means
    clustering.

    Parameters
    ----------
    y_values : array-like
        1D array of y-values (peak heights).
    n_clusters : int, optional
        Number of clusters to form, by default 2 (noise vs signal).

    Returns
    -------
    float
        Threshold value separating noise from peaks.
    """
```

```
    if len(y_values) < 2:
        print("Need at least two data points to apply clustering.")
        return None
    y_values = np.array(y_values).reshape(-1, 1)
    labels, centroids = kmeans_clustering(y_values, n_clusters)
    threshold = np.sort(centroids.ravel())[0]
    return threshold

def find_peaks(y, height):
    """
    Identify peaks in a 1D data array using local maxima detection.

    Parameters
    ----------
    y : numpy.ndarray
        Input data array.
    min_height : float
        Minimum height for peak detection.

    Returns
    -------
    list
        Indices of detected peaks.
    """

    peaks = [i for i in range(1, len(y) - 1)
             if y[i] > height and y[i] > y[i - 1] and y[i] > y[i + 1]]

    return peaks

def detect_and_plot_peaks(x, y, threshold_baseline, save_path, xlim=None,
clusters=None):
    """
    Detects and plots peaks in the given data with a wide, detailed NMR
    spectrum style.
    Optionally highlights peaks in different colours based on clusters.

    Parameters
    ----------
    x : numpy.ndarray
        X-axis data (Chemical shift in ppm).
    y : numpy.ndarray
        Y-axis data (Intensity).
    threshold_baseline : float
        Minimum height for peak detection.
    save_path : path
        Path to save plot image.
    xlim : tuple of (float, float), optional
        X-axis limits as (min, max). Defaults to the full range of x data.
```

```
    clusters : dict, optional
        A dictionary where keys are cluster labels and values are lists of
        peak values.
        Peaks in different clusters are plotted in different colours.

    Returns
    -------
    peak_x : numpy.ndarray
        X coordinates of detected peaks.
    peak_y : numpy.ndarray
        Y coordinates of detected peaks.
    y_axis_scale : int
        The exponent of the scientific notation scale for the y-axis (e.g.,
        -3 if the axis shows  10 ^-3).
    """
    xhigh, xlow = max(x), min(x)
    peaks = find_peaks(y, height=threshold_baseline)
    peak_x, peak_y = x[peaks], y[peaks]

    plt.figure(figsize=(12, 8))
    plt.plot(x, y, linestyle="-", linewidth=0.8, color="black", label="NMR
    Spectrum")
    plt.axhline(y=threshold_baseline, color='gray', linestyle="--",
    linewidth=1, label="Threshold")

    # Default colour map for clusters
    colours = plt.cm.get_cmap('tab10', len(clusters) if clusters else 1)

    if clusters:
        for i, (label, peak_values) in enumerate(clusters.items()):
            mask = np.isin(peak_x, peak_values)
            cluster_x = peak_x[mask]
            cluster_y = peak_y[mask]
            plt.scatter(cluster_x, cluster_y, color=colours(i), marker="x",
            s=50)
    else:
        plt.scatter(peak_x, peak_y, color="red", marker="x", s=50, label="
        Detected Peaks")

    if xlim:
        plt.xlim(xlim[1], xlim[0])
    else:
        plt.xlim(xhigh, xlow)

    plt.xlabel("Chemical Shift (ppm)", fontsize=12)
    plt.ylabel("Intensity", fontsize=12)
    plt.title(" H  NMR Spectrum", fontsize=14)
    if not clusters:
        plt.legend(loc="upper right", fontsize=10)
```

```
    plt.savefig(save_path, dpi=300, bbox_inches='tight')
    plt.show(block=False)

     # Get the y-axis scaling from the scientific notation
    ax = plt.gca()
    offset_text = ax.yaxis.get_offset_text().get_text()

    # Simple extraction of the exponent from scientific notation like "1e-3"
    try:
        _, exponent = offset_text.split('e')
        y_axis_scale = int(exponent)
    except ValueError:
        y_axis_scale = 0  # No scientific notation used

    return peak_x, peak_y, y_axis_scale

def calculate_cluster_count(peaks):
    """
    Estimate the number of clusters in peak data using K-Means on peak
    distances.

    Parameters
    ----------
    peaks : numpy.ndarray
        Sorted array of peak positions.

    Returns
    -------
    count: int
        Estimated count of clusters based on distance thresholding.
    """
    peaks = np.sort(np.asarray(peaks, dtype=float))

    # Handle case where there's only two or less peaks
    if len(peaks) < 3:
        return len(peaks)

    distances = np.diff(peaks)
    labels, centroids = kmeans_clustering(distances.reshape(-1, 1), k=2)
    threshold = np.min(centroids)
    count = np.sum(distances > threshold)

    return int(count)

def assign_functional_groups(peak):
    """
    Determine possible functional groups based on a chemical shift value.
```

```
    Parameters
    ----------
    peak : float
        Chemical shift value to classify.

    Returns
    -------
    list
        List of possible functional groups corresponding to the chemical
        shift.
        Returns ["Unassigned"] if no match is found.
    """
    possible_groups = [group for group, (low, high) in CHEMICAL_SHIFT_RANGES
    .items() if low <= peak <= high]
    return possible_groups if possible_groups else ["Unassigned"]
```

j_val.py

```
import numpy as np
from sklearn.cluster import DBSCAN

def factorize_multiplicity(num_peaks: int) -> list[str]:
    """
    Algorithm to generate all possible multiplicity labels for a given
    number of peaks using backtracking.

    Parameters
    ----------
    num_peaks : int
        The number of peaks to factorize.

    Returns
    -------
    list of str
        A sorted list of possible multiplicity labels or ["multiple"] if no
        valid factorisation exists.

    Algorithm
    ---------
    This function uses a backtracking algorithm to explore all possible
    factorizations of `num_peaks`
    using specific factor ranges and maps these factorizations to their
    respective labels.

    Rules
    -----
    - If num_peaks > 36, returns ["multiplet"].
    - If num_peaks == 1, returns ["s"] (singlet).
    - If num_peaks <= 7:
```

```
    - Uses possible factors [2..7] with specific labels.
    - Generates all permutations of valid factorizations.
- If num_peaks > 7:
    - Restricts factors to [2, 3, 4] with labels ("d", "t", "q").
    - Returns ["multiplet"] if no valid factorization is found.

Examples
--------
>>> factorize_multiplicity(4)
['dd', 'q']
>>> factorize_multiplicity(8)
['ddd', 'dq', 'qd']
>>> factorize_multiplicity(1)
['s']
>>> factorize_multiplicity(40)
['multiplet']
>>> factorize_multiplicity(11)
['multiplet']
"""
if num_peaks > 36:
    return ["multiplet"]
if num_peaks == 1:
    return ["s"]

FACTORS_UP_TO_7 = {
    2: "d",
    3: "t",
    4: "q",
    5: "quintet",
    6: "hex",
    7: "hept"
}
FACTORS_ABOVE_7 = {
    2: "d",
    3: "t",
    4: "q"
}

if num_peaks <= 7:
    factor_dict = FACTORS_UP_TO_7
    factor_range = range(2, 8)
else:
    factor_dict = FACTORS_ABOVE_7
    factor_range = range(2, 5)

results = []

def backtrack(current, path):
    if current == 1:
```

```
                results.append(path[:])
                return
            for f in factor_range:
                if current % f == 0:
                    path.append(f)
                    backtrack(current // f, path)
                    path.pop()

    backtrack(num_peaks, [])

    labels = set()
    for seq in results:
        label_str = "".join(factor_dict[f] for f in seq)
        labels.add(label_str)

    if num_peaks <= 7 and num_peaks in factor_dict:
        labels.add(factor_dict[num_peaks])

    final_labels = sorted(labels)

    return final_labels if final_labels else ["multiplet"]

def multiplicity_to_line_count(multiplicity):
    """
    Convert multiplicity notation ('d', 't', 'q', 'quintet', 'hept') to the
    corresponding line counts.

    Parameters
    ----------
    multiplicity : str
        Multiplicity indicator (e.g., 'd', 't', 'quintet', 'hept').

    Returns
    -------
    list of int
        Line counts for each component of the multiplicity.

    Examples
    --------
    >>> multiplicity_to_line_count('td')
    [3, 2]

    """
    lookup = {'d': 2, 't': 3, 'q': 4, 'quintet': 5, 'hex': 6, 'hept': 7}
    if multiplicity.lower() in lookup:
        return [lookup[multiplicity.lower()]]
    return [lookup.get(char.lower()) for char in multiplicity]
```

```
def generate_splitting_path(dimensions):
    """
    Recursively generate the splitting path for given multiplicity
    dimensions.

    Parameters
    ----------
    dimensions : list of int
        Number of lines in each splitting dimension (e.g., [2, 3] for a
        doublet-triplet).

    Yields
    ------
    tuple of int
        Indexes representing the splitting pattern path.

    Algorithm:
    ----------
    1. Generate the path for the innermost dimension as sequential indices.
    2. For each additional dimension:
       - Combine each index of the current dimension with all indices of the
        existing path.
       - Alternate the direction of the index combination for every row.
       - Append the resulting combinations to the path.
    3. The final path maintains a logical order where each step only changes
     a single index.

    Example:
    --------
    For dimensions = [2, 3] (representing a doublet-triplet):

    list(generate_splitting_path([2, 3]))

    Output:
    [(0, 0), (0, 1), (0, 2),
     (1, 2), (1, 1), (1, 0)]
    """
    if not dimensions:
        return
    if len(dimensions) == 1:
        for i in range(dimensions[0]):
            yield (i,)
    else:
        first_dim = dimensions[0]
        remaining_dims = dimensions[1:]
        subpath = list(generate_splitting_path(remaining_dims))
        reversed_subpath = subpath[::-1]
        for i in range(first_dim):
            if i % 2 == 0:
```

```
                for combo in subpath:
                    yield (i,) + combo
            else:
                for combo in reversed_subpath:
                    yield (i,) + combo


def generate_j_pattern(multiplicity):
    """
    Generate the expected rank order of J values for a given multiplicity
    pattern.

    This function translates a multiplicity string (e.g., 'dt' for doublet-
    triplet)
    into line counts using `multiplicity_to_line_count` and generates the
    splitting
    path using `generate_splitting_path`. It then evaluates which dimension
    changes
    at each step to assign ranks, with higher ranks assigned to higher
    dimension changes.

    Parameters
    ----------
    multiplicity : str
        Multiplicity pattern (e.g., 'td', 'hept', etc.).

    Returns
    -------
    list of int
        Rank order of J values from largest to smallest.

    Examples
    --------
    >>> generate_j_pattern('dt')
    [1, 1, 2, 1, 1]

    Notes
    -----
    The function generates the splitting path as:

    [(0, 0), (0, 1), (0, 2),
     (1, 2), (1, 1), (1, 0)]

    Rank Assignment:
    - (0, 0) to (0, 1): Second dimension changes     Rank 1
    - (0, 1) to (0, 2): Second dimension changes     Rank 1
    - (0, 2) to (1, 2): First dimension changes     Rank 2
    - (1, 2) to (1, 1): Second dimension changes     Rank 1
    - (1, 1) to (1, 0): Second dimension changes     Rank 1
```

```
    Resulting in the rank order:
    [1, 1, 2, 1, 1]
    """

    sizes = multiplicity_to_line_count(multiplicity)
    dimension_count = len(sizes)
    path = list(generate_splitting_path(sizes))
    ranks = []
    for p1, p2 in zip(path, path[1:]):
        for dim in range(dimension_count):
            if p1[dim] != p2[dim]:
                ranks.append(dimension_count - dim)
                break
    return ranks

def cluster_and_rank_j_values(j_values, uncertainty=1.0):
    """
    Cluster J values based on uncertainty and output a ranked list of
    clusters.

    Parameters
    ----------
    j_values : list of float
        J coupling values in Hz.
    uncertainty : float, optional
        Clustering threshold in Hz, default is 1.0 Hz.

    Returns
    -------
    list of int
        Ranked J values based on cluster ordering.

    Examples
    --------
    >>> cluster_and_rank_j_values([7.1, 7.1, 9], uncertainty=1)
    [1, 1, 2]

    Notes
    -----
    Clusters J values using DBSCAN with the specified uncertainty as the
    clustering radius.
    Outputs a ranked list of clusters based on ascending mean J values.
    """
    j_values_reshaped = np.array(j_values).reshape(-1, 1)
    clustering = DBSCAN(eps=uncertainty, min_samples=1).fit(
    j_values_reshaped)
    labels = clustering.labels_
    unique_labels = np.unique(labels)
```

```
    sorted_clusters = sorted(unique_labels, key=lambda lbl: np.mean(
    j_values_reshaped[labels == lbl]))
    rank_map = {label: rank + 1 for rank, label in enumerate(sorted_clusters
    )}
    return [rank_map[label] for label in labels]


def match_peaks_to_multiplicity(peaks, multiplicities, frequency,
uncertainty=1.0):
    """
    Matches observed peak positions to an expected multiplicity pattern.

    Parameters
    ----------
    peaks : list of float
        Observed peak positions (ppm).
    multiplicities : list of str
        Candidate multiplicity patterns (e.g., ['d', 't', 'td']).
    frequency : float
        Spectrometer frequency in MHz (for  p p m Hz  conversion).
    uncertainty : float, optional
        Threshold (Hz) for clustering J-values.

    Returns
    -------
    matched_multiplicity : str
        Identified pattern (e.g., 'd', 'dt') if exactly one match, otherwise
         'multiplet'.
    avg_j_by_rank : list of float or None
        Averaged J-values by rank if one pattern matches.
    raw_j_vals : list of float or None
        Original J-values if one pattern matches, otherwise adjacent J-
        values.

    Notes
    -----
    - If the first item in `multiplicities` is 'multiplet', we immediately
    return
      ('multiplet', <adjacent_Js>).
    - Otherwise, we test each multiplicity pattern by:
        1. Calculating J values from the peaks.
        2. Clustering and ranking them with `cluster_and_rank_j_values`.
        3. Comparing that rank-pattern to `generate_j_pattern(multiplicity)
        `.
      If exactly one matches, we return it. If 0 or >1 match, we label '
      multiplet'.
    """

    if not multiplicities or multiplicities[0].lower() == 'multiplet':
```

```
        return 'multiplet', '', _adjacent_j_values(peaks, frequency)

    matched_patterns = []
    stored_j_vals_and_ranks = []

    for m in multiplicities:
        j_vals = calculate_j_vals(peaks, m, frequency)
        j_ranks = cluster_and_rank_j_values(j_vals, uncertainty)
        if j_ranks == generate_j_pattern(m):
            matched_patterns.append(m)
            stored_j_vals_and_ranks.append((m, j_vals, j_ranks))

    if len(matched_patterns) == 1:
        mpat, j_vals, j_ranks = stored_j_vals_and_ranks[0]
        avg_j_by_rank = [np.mean([j for j, rk in zip(j_vals, j_ranks) if rk
        == ur])
                         for ur in np.unique(j_ranks)]
        return mpat, avg_j_by_rank, j_vals

    return 'multiplet', '', _adjacent_j_values(peaks, frequency)


def _adjacent_j_values(peaks, frequency):
    """
    Helper function to convert consecutive peak differences (in ppm) into Hz
    .
    """
    diffs_ppm = np.diff(peaks)
    diffs_hz = [d * frequency for d in diffs_ppm]
    return diffs_hz


def calculate_j_vals(peaks, multiplicity, frequency):
    """
    Calculates J-values (Hz) for a given set of NMR peaks and multiplicity.

    Parameters
    ----------
    peaks : array-like
        Chemical shift peak positions (ppm).
    multiplicity : str
        Multiplicity notation ('d', 't', 'q', 'dt', etc.).
    frequency : float
        Spectrometer frequency (MHz), used to convert ppm differences to Hz.

    Returns
    -------
    list of float
        J-values (Hz) ordered according to the multiplicity pattern.
```

```
    Notes
    -----
    - If a single-line pattern (e.g., singlet, doublet), returns adjacent J-
    values.
    - Otherwise, iteratively:
        1. Groups peaks based on multiplicity.
        2. Computes J-values from peak differences.
        3. Places J-values according to rank pattern.
        4. Replaces peak groups with their mean for the next iteration.
    """

    line_counts = multiplicity_to_line_count(multiplicity)

    if len(line_counts) == 1:
        return _adjacent_j_values(peaks, frequency)

    line_counts_reversed = line_counts[::-1]
    current_peaks = peaks.copy()
    rank_pattern = generate_j_pattern(multiplicity)
    final_j_vals = [0] * len(rank_pattern)

    for iteration, n in enumerate(line_counts_reversed):
        rank_number = iteration + 1
        n_groups = len(current_peaks) // n
        groups = [current_peaks[i * n:(i + 1) * n] for i in range(n_groups)]

        # Compute J-values (Hz) from peak differences within each subgroup
        j_vals_iteration = [(np.diff(g) * frequency).tolist() for g in
        groups]
        j_vals_iteration = [j for sublist in j_vals_iteration for j in
        sublist]

        # Assign J-values to correct positions based on rank pattern
        indices_to_fill = [i for i, rank in enumerate(rank_pattern) if rank
        == rank_number]
        for idx, val in zip(indices_to_fill, j_vals_iteration):
            final_j_vals[idx] = val

        # Replace each subgroup with its mean to form new representative
        peaks
        current_peaks = np.array([np.mean(g) for g in groups])

    return final_j_vals
```

main.py

```
import os
import argparse
```

```
from pathlib import Path

from cluster import (
    hierarchical_clustering,
    assign_functional_groups,
    load_csv_coordinates,
    determine_noise_threshold,
    detect_and_plot_peaks,
    calculate_cluster_count,
)

from j_val import (
    factorize_multiplicity,
    match_peaks_to_multiplicity,
)

def nmr_peak_analysis(peaks, k, frequency, uncertainty, extra, base_name,
output_dir):
    """
    Clusters NMR peaks, assigns functional groups, classifies multiplets,
    and
    prints and saves peaks with multiplicity and J values in the standard
    NMR format.
    Example output:    6.84 ppm (d, J = 2.09 Hz)
    """

    detailed_path = os.path.join(output_dir, f"{base_name}
    _detailed_calculation.txt")
    simple_path = os.path.join(output_dir, f"{base_name}_simple_calculation.
    txt")

    for path in [detailed_path, simple_path]:
        if os.path.exists(path):
            os.remove(path)

    if k > len(peaks): # Potential error when k is input again by user.
        raise ValueError("K is higher than the number of peaks available.")

    max_k = min(k + extra, len(peaks)) # In the case extra leads to more k
    than peaks
    min_multiplets, best_k = float('inf'), k

    for current_k in range(k, max_k + 1):
        multiplet_count, clusters, cluster_data = evaluate_clusters(peaks,
        current_k, uncertainty, frequency)
        if multiplet_count < min_multiplets:
            min_multiplets, best_k = multiplet_count, current_k
            best_cluster_data = cluster_data
            best_clusters = clusters
```

```
    # Order all data by peak values
    best_cluster_data.sort(key=lambda x: min(x[0]))

    header = f" H  NMR ({int(round(frequency))} MHz)"
    print(f"{header}")

    with open(detailed_path, "a") as f_detailed, open(simple_path, "a") as
    f_simple:

        f_detailed.write(f"{header}\n")
        f_simple.write(f"{header}\n")

        for cluster_peaks, best_match, avg_j_values, assigned_groups,
        all_multiplicities, all_j_vals in best_cluster_data:
            write_detailed_output(f_detailed, cluster_peaks, all_j_vals,
            assigned_groups, all_multiplicities)
            write_simple_output(f_simple, cluster_peaks, best_match,
            avg_j_values, assigned_groups)

            if best_match == "multiplet":
                range_str = f"{min(cluster_peaks):.2f}-{max(cluster_peaks)
                :.2f}"
            else:
                range_str = f"{min(cluster_peaks):.2f}"

            if avg_j_values:
                j_values_str = ', '.join(f'{j:.2f}' for j in avg_j_values)
                peak_str = f"   {range_str} ppm ({best_match}, J = {
                j_values_str} Hz)"
            else:
                peak_str = f"   {range_str} ppm ({best_match})"

            print(peak_str)

    print(f"Analysis complete. Files written to: {detailed_path}, {
    simple_path}")
    return best_k, best_clusters


def evaluate_clusters(peaks, k, uncertainty, frequency):
    """
    Performs hierarchical clustering on NMR peaks, assigns multiplicities
    and functional groups
    to each cluster, and returns the number of detected multiplets, clusters
    , and analysed cluster data.
    """

    labels, centroids = hierarchical_clustering(peaks.reshape(-1, 1), k)
```

```
    clusters = {i: [] for i in range(k)}
    cluster_data = []
    multiplet_count = 0
    for peak, label in zip(peaks, labels):
        clusters[label].append(peak)
    for cluster_peaks in clusters.values():
        cluster_peaks.sort()
        best_match, avg_j_values, assigned_groups, all_multiplicities,
        all_j_vals = process_cluster(cluster_peaks, uncertainty, frequency)
        cluster_data.append((cluster_peaks, best_match, avg_j_values,
        assigned_groups, all_multiplicities, all_j_vals))
        if best_match == 'multiplet':
            multiplet_count += 1
    return multiplet_count, clusters, cluster_data


def process_cluster(cluster_peaks, uncertainty, frequency):
    """
    Processes a single cluster of peaks to determine the best multiplicity
    match,
    coupling constants (J values), and assigned functional groups.
    """
    if len(cluster_peaks) == 1:
        return "s", [], [], [], []

    all_mults = factorize_multiplicity(len(cluster_peaks))
    best_match, matched_j_vals, raw_j_vals = match_peaks_to_multiplicity(
        peaks=cluster_peaks,
        multiplicities=all_mults,
        frequency=frequency,
        uncertainty=uncertainty
    )

    assigned_groups = assign_functional_groups(cluster_peaks[0])

    return best_match, matched_j_vals, assigned_groups, all_mults,
    raw_j_vals

def write_detailed_output(f, peaks, j_vals, groups, all_multiplicities):
    peaks_str = ', '.join(f"{p:.5f}" for p in peaks)
    j_vals_str = ', '.join(f"{j:.5f}" for j in j_vals) if j_vals else None
    groups_str = ", ".join(groups) if groups else "Not Detected"
    mult_str = ', '.join(str(m) for m in all_multiplicities) if
    all_multiplicities else "Not Detected"
    f.write("========================================\n")
    f.write(f"Chemical Shifts (ppm): {peaks_str}\n")
    if j_vals_str:
        f.write(f"J-values (Hz): {j_vals_str}\n")
    f.write(f"Possible Groups: {groups_str}\n")
```

```
    f.write(f"All Identified Multiplicities: {mult_str}\n\n")

def write_simple_output(f, peaks, best_match, j_vals, groups):
    if best_match == "multiplet":
        range_str = f"{min(peaks):.2f}-{max(peaks):.2f}"
    else:
        range_str = f"{peaks[0]:.2f}"
    j_vals_str = ', '.join(f"{j:.2f}" for j in j_vals) if j_vals else None
    groups_str = ", ".join(groups) if groups else "Not Detected"
    if j_vals_str:
        f.write(f"   {range_str} ppm ({best_match}, J = {j_vals_str} Hz,
        possible groups: {groups_str})\n")
    else:
        f.write(f"   {range_str} ppm ({best_match}, possible groups: {
        groups_str})\n")

def main(path, frequency, uncertainty, extra):
    """Main function to perform NMR peak analysis with adjustable parameters
    ."""
    x, y = load_csv_coordinates(path)
    noise_threshold = determine_noise_threshold(y)
    output_dir = 'nmrdata_output'
    os.makedirs(output_dir, exist_ok=True)

    filename = os.path.basename(path)
    base_name = os.path.splitext(filename)[0]
    save_path = os.path.join(output_dir, f"{base_name}_nmr_plot.jpg")

    # Peak Detection and Plotting
    while True:
        peak_x, peak_y, y_axis_scale = detect_and_plot_peaks(x, y,
        threshold_baseline=noise_threshold, save_path=save_path)
        scaled_threshold = noise_threshold / (10 ** y_axis_scale)

        print(f"Current threshold: {scaled_threshold:.2f}  10 ^{y_axis_scale
        }")

        if len(peak_x) == 0: # In case user puts threshold too high and no
        peaks select.
            print('No peaks detected, try another threshold.')
        elif input("Is this threshold acceptable? (y/n): ").strip().lower()
        == 'y':
            break

        user_input_str = input(f"Enter new threshold (as seen on y-axis, e.g
        . 4.5 if axis shows 4.5  10 ^{y_axis_scale}): ")
        try:
            user_input_value = float(user_input_str)
            noise_threshold = user_input_value * (10 ** y_axis_scale)
```

```
            except ValueError:
                print("Invalid input; please enter a numeric value.")
                continue

        # Clustering and Analysis
        k = calculate_cluster_count(peak_x)

        while True:
            k, clusters = nmr_peak_analysis(peak_x, k, frequency=frequency,
            uncertainty=uncertainty, extra=extra, base_name=base_name,
            output_dir=output_dir)
            detect_and_plot_peaks(x, y, threshold_baseline=noise_threshold,
            clusters=clusters, save_path=save_path)
            print(f"Number of clusters: {k}")

            if input("Is this number of clusters acceptable? (y/n): ").strip().
            lower() == 'y':
                break

            try:
                k = int(input("Enter preferred number of clusters: "))
                extra = 0
            except ValueError:
                print("Invalid input; please enter an integer.")
                continue

    def validate_file(arg):
        if Path(arg).is_file():
            return arg
        raise FileNotFoundError(arg)

    if __name__ == "__main__":
        parser = argparse.ArgumentParser(description="NMR Peak Analysis Tool")
        parser.add_argument("--file", type=validate_file, required=True, help="
        Input file path")
        parser.add_argument("--extra", type=int, default=0, help="Extra clusters
         to consider during analysis")
        parser.add_argument("--uncertainty", type=float, default=1.0, help="
        Uncertainty tolerance for multiplicity matching")
        parser.add_argument("--frequency", type=float, required=True, help="NMR
        frequency (optional, default is 400.0 MHz)")
        args = parser.parse_args()
        main(args.file, uncertainty=args.uncertainty, extra=args.extra,
        frequency=args.frequency)
```

References

[1] Andel. ^{1}H-nmr spectrum of ethanol. https://en.m.wikipedia.org/wiki/File:1H_NMR_Ethanol_Coupling_shown.svg, 2019. Plotted using Microsoft Excel, converted with Inkscape, and edited in Atom. Data sourced from SDBSWeb (https://sdbs.db.aist.go.jp/sdbs/cgi-bin/landingpage?sdbsno=1300), National Institute of Advanced Industrial Science and Technology. Accessed 2019-08-03.

[2] Steven A. Hardinger. Illustrated glossary of organic chemistry: Triplet of doublets. https://www.chem.ucla.edu/~harding/IGOC/T/triplet_of_doublets.html, 2010–2017. Department of Chemistry & Biochemistry, UCLA. All rights reserved. Some images used with permission; others are original or public domain. Accessed: 2025-03-26.

[3] Giulia Fischetti, Nicolas Schmid, Simon Bruderer, Björn Heitmann, Andreas Henrici, Alessandro Scarso, Guido Caldarelli, and Dirk Wilhelm. A deep learning framework for multiplet splitting classification in 1h nmr. *Journal of Magnetic Resonance*, 373:107851, April 2025.

[4] Zhi Liu, Ahmed Abbas, Bing-Yi Jing, and Xin Gao. Wavpeak: picking nmr peaks through wavelet-based smoothing and volume-based filtering. *Bioinformatics*, 28(7):914–920, February 2012.

[5] Giulia Fischetti, Nicolas Schmid, Simon Bruderer, Guido Caldarelli, Alessandro Scarso, Andreas Henrici, and Dirk Wilhelm. Automatic classification of signal regions in 1h nuclear magnetic resonance spectra. *Frontiers in Artificial Intelligence*, 5, January 2023.

[6] Hmdb spectrum for metabolite (test 2). https://hmdb.ca/spectra/nmr_one_d/1120. Accessed: 27 March 2025.

[7] Hmdb spectrum for metabolite (test 4). https://hmdb.ca/spectra/nmr_one_d/1070. Accessed: 27 March 2025.

[8] Hmdb spectrum for metabolite (test 3). https://hmdb.ca/spectra/nmr_one_d/5245. Accessed: 27 March 2025.

[9] Hmdb spectrum for metabolite (test 6). https://hmdb.ca/spectra/nmr_one_d/1122. Accessed: 27 March 2025.

[10] Hmdb spectrum for metabolite (test 1). https://hmdb.ca/spectra/nmr_one_d/1043. Accessed: 27 March 2025.

[11] Hmdb spectrum for metabolite (test 5). https://hmdb.ca/spectra/nmr_one_d/1078. Accessed: 27 March 2025.