# Molecular Matching with Ion Mobility Spectrometry using Python

## Introduction

The idea of this research is to analyze and identify how useful the Python programming language can be in comparing and identifying substances using Ion Mobility Spectrometry. Ion Mobility Spectrometry is an analytical research method that helps in separating and identifying ionized molecules.

Ion Mobility Spectrometry (IMS) is widely used in identification of molecules in fields like security, military, research etc. and is extensively used in detection of explosives, drugs and chemical weapons.

This research explores the idea of a Python-based workflow for molecular matching with IMS Data with the following steps

1. Data retrieval from an SQLite database:

The SQLite database should have two tables, *measurement* and *library* where the measurement table holds the IMS Data with which we will be working on identifying the relevant data that we need to compare and identify the substance.

1. Calculation of reduced ion mobility:

Reduced ion mobility is an important parameter with which each molecule can be differentiated.

1. Library matching:

Compare the calculated reduced ion mobility values against a reference database (which should be the library table from the SQLite database) and assign the most likely molecular identity.

## Background and Theoretical Framework

### 2.1 What is Ion Mobility Spectrometry

According to James N Dodds and Erin S Baker, Ion Mobility Spectrometry (IMS) is the study of how ions move in gases under the influence of an electric field. The idea is the setup consists of multiple chambers.[[1]](#footnote-1) When we consider a rudimentary setup of an IMS, the first chamber is where the sample gas enters. Then the sample gas is ionized using an external source by exciting the molecules. Then the ionized gas enters another chamber which has an electric field. Each molecule has a different *drift velocity* with which they travel this chamber from start to end. We should know the voltage of this electric field, the temperature, pressure and multiple factors that are required to calculate the reduced ion mobility ion mobility of each molecule in that sample.

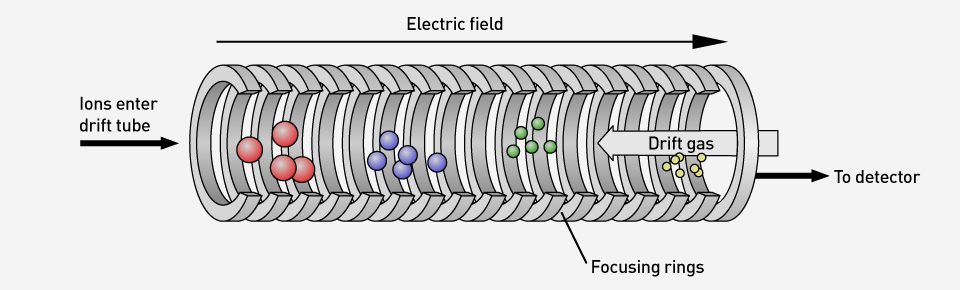


Figure Ion Mobility Spectrometry cross-section [[2]](#footnote-2)

### Principles of Ion Mobility Spectrometry

In IMS, ions that are generated from a sample are introduced in the drift region. In a uniform electric field, each ion’s drift velocity is proportional to the electric field strength . is the proportionality factor between the drift velocity and the electric field strength.[[3]](#footnote-3)

The reduced ion mobility is calculated when other factors such as the temperature and pressure during drift, standard temperature *T*0 = 273 K and standard pressure *p*0 = 1013 hPa are known.

To find the value, we can use the following formula.

Where is the drift length (the length of the chamber or tube where the ionized particles move), is the drift time and is the potential difference in the chamber.

### 2.3 Molecular Matching using

Molecular matching values is straightforward. We get the value of each spectrum using the above-mentioned calculations. We match the values against a reference library. is the value calculated. For each measured value , the algorithm searches for the library entry that minimizes . Each substance in the library is represented by up to 3 positive-mode and 3 negative-mode entries, which shows slight experimental variations. A simple molecular matching absolute-deviation criterion will be used to assign the *closest* substance to each measurement.

## 3. Python for Molecular Matching

Python can be an extremely effective tool in the process of molecular matching substances in Ion Mobility Spectrometry. There are multiple reasons to use Python as the main tool for the procedure.

### 3.1 File Parsing & Data Handling

* It is possible that the values necessary for the calculation of values may not necessarily be from an SQL database. Some IMS devices have serial communication capabilities with their own APIs. Data can be stored in files which can be simple text *.txt* or *.csv* or sometimes *JSON* files. Python has a vast set of libraries for parsing different types of data and it is extremely reliable as well.
* *Pandas* *DataFrames* are very efficient in converting data into tabular form providing indexing, filtering, grouping and even being part of the I/O (CSV <-> DataFrame <-> SQL) with very few lines of code. Pandas is widely used for data handling in Python.

### 3.2 Scientific & Statistical Computation

* Mathematical calculations and operations can be done fairly easily using the Python numeric stack. Some libraries like the *math* library and *numpy* library lets us implement a lot of formulas easily and can be used to make hundreds or thousands of measurements very quickly.

### 3.3 Database Integration

* For our project, database integration helps us a lot by defining tables in the *SQLite* Database which can be easily connected to our project using very good libraries. One such library is the *SQLAlchemy* library which helps us in defining the tables that are necessary to be read and or updated in python classes.
* Portability is also a very important advantage of using Python’s database integration. Switching between SQLite to PostgreSQL or MySQL or almost any other fairly known and commonly used database frameworks can be done with minor tweaks and corrections.

### 3.4 Visualization

* Python Visualization frameworks are well-known for easy use and understanding. Plotting drift-time distributions, trends or the spectra in itself can be fairly easily done. Frameworks like *MatPlotLib, Bokeh, Seaborn* can be used to make time-series of identifications or an interactive Bokeh Plot for matching errors.

### 3.5 Reproducability & Testing

* Unit Testing is an important aspect of programming any project. *PyTest* or *unittest* frameworks are really good at writing unit tests to test the project.
  + Parsing of header sections can be tested.
  + Peak-finding logic can be tested for both the positive and the negative spectra.
  + By providing known values, accurate values can be calculated and tested.
  + Library-matching logic can be tested using edge conditions (values that are close to matching but no matches).

## 4. Computational Tools: Python Ecosystem

Since Python offers such a rich ecosystem for some instrument control and data science, we will be able to do the following methodologies in finding the values of each spectrum.

* File Parsing: Existing data if it is in *csv* format or any other format, we can parse the code using modules/libraries like *re*, *numpy* and *csv.* Retrieving data such as raw channel[[4]](#footnote-4) values, IMS parameters and arrays.
* Numerical Computation: For numerical computation, we can either write our own functions which calculate the necessary value using the formulas mentioned earlier. Or for some other operations like drift-time calculation, *numpy* is a very good tool in handling such operations. *Pandas*  can be used to provide us the tabular data management.
* Databases: *SQLAlchemy* is used along with *SQLite* which enable structured data storage of raw measurements and the reference libraries.
* Visualization: *Matplotlib* helps in visualising the spectrum, maybe after Savitzky–Golay[[5]](#footnote-5) filtration and after finding the necessary peaks for which the values are to be calculated.

## 5. Programming

Molecular matching using values is done in the following steps.

1. Raw spectrum data that is received from the sensor, meta data such as temperature, pressure, drift tube length etc. are necessary to calculate values. These values are usually sent by the IMS device used. For example, [AIRSENSE Analytics GmbH](https://airsense.com/en) from Schwerin Germany, work extensively with IMS devices and they send the data using serial communication over RS-232 connection. If the necessary API is available, the necessary data can be parsed.
2. The Raw spectrum is then parsed and saved in an SQLite database. The database has two tables, *measurements* and *library.* The measurements table holds the necessary data that is required to calculate the values and library table holds existing substances with known values for reference.
3. A module is created that will handle all SQL related functions that can be called to get the relevant data for calculation.
4. Another module called *ims.py* is created to handle all IMS calculations and algorithms.
5. *Scipy* module is used to filter noises from the raw spectrum and can be used to find peaks from the spectrum. It is not necessary to calculate value of every single point on the spectrum.

### 5.1 Raw Data Handling

Firstly the *sqlite\_helper.py* module is created which will be handling any SQLite database related functionalities. The measurements table is created using the following SQL Query.

1. CREATE TABLE measurements (

2. id INTEGER PRIMARY KEY AUTOINCREMENT,

3. measurement\_time TEXT,

4. channel\_1 REAL,

5. channel\_2 REAL,

6. channel\_3 REAL,

7. channel\_4 REAL,

8. channel\_5 REAL,

9. channel\_6 REAL,

10. channel\_7 REAL,

11. channel\_8 REAL,

12. dilution REAL,

13. temperature\_drift\_tube REAL,

14. pressure REAL,

15. pos\_voltage REAL,

16. neg\_voltage REAL,

17. tube\_length REAL,

18. pressure\_offset REAL,

19. pressure\_gradient REAL,

20. pos\_spectrum BLOB,

21. neg\_spectrum BLOB

22. );

23.

This query creates the measurement table with the important parameters like measurement time, temperature of the drift tube during measurement, pressure during measurement, potential difference during measurement (voltage), tube length and the positive and negative spectra.

Similarly, the library table is created with the following query.

CREATE TABLE library (

    id INTEGER PRIMARY KEY AUTOINCREMENT,

    substance\_name TEXT,

    k0\_pos\_1 REAL,

    k0\_pos\_2 REAL,

    k0\_pos\_3 REAL,

    k0\_neg\_1 REAL,

    k0\_neg\_2 REAL,

    k0\_neg\_3 REAL

);

A class is created in python to handle the measurement table since this table contains a bit more data compared to the library table. The class *Measurement* in the *sqlite\_helper.py* module helps us in having an idea to understand how the measurement table is structured.

### 5.2 Substance Identification

The module *substance\_identifier.py* has the methods *identify\_substances* which helps us in the comparison of the values calculated from the spectrum to the values from the library table.

The identification process is as follows:

1. Library retrieval:
   1. The function retrieves the substance library from the database using the *sqlite\_helper.get\_substance\_library()* function.
   2. The library contains the substance names and their reference values (3 positive and 3 negative).
2. Matching Algorithm:
   1. For each substance in the library the function checks if the measured positive values match the reference values within a tolerance of 0.2 (usually the tolerance is around 0.02, but for the precision, there are more compensations that must be done like temperature compensation, pressure, humidity etc. For the sake of simplicity, 0.2 as tolerance has been selected)
   2. Similarly, the negative values are matched and only non-zero reference are considered for matching.
3. Match criteria:
   1. All required positive values should match (if any specified in the library), OR
   2. All required negative values match (if any specified in the library)
4. Results collection:
   1. For each identified substance, the function collects the substance name, the positive spectrum matches, and negative spectrum matches.

### 5.3 Visualization

The *visualization.py* module helps in displaying IMS spectrum data with some interactive features.

There are two key functions in this module.

1. create\_spectrum\_plot:

Creates a static plot showing the positive and negative spectra with the peaks being annotated. The function returns a matplotlib figure object.

1. Show\_scrollable\_plots:

Creates an interactive visualization with a slider to navigate through multiple measurements.

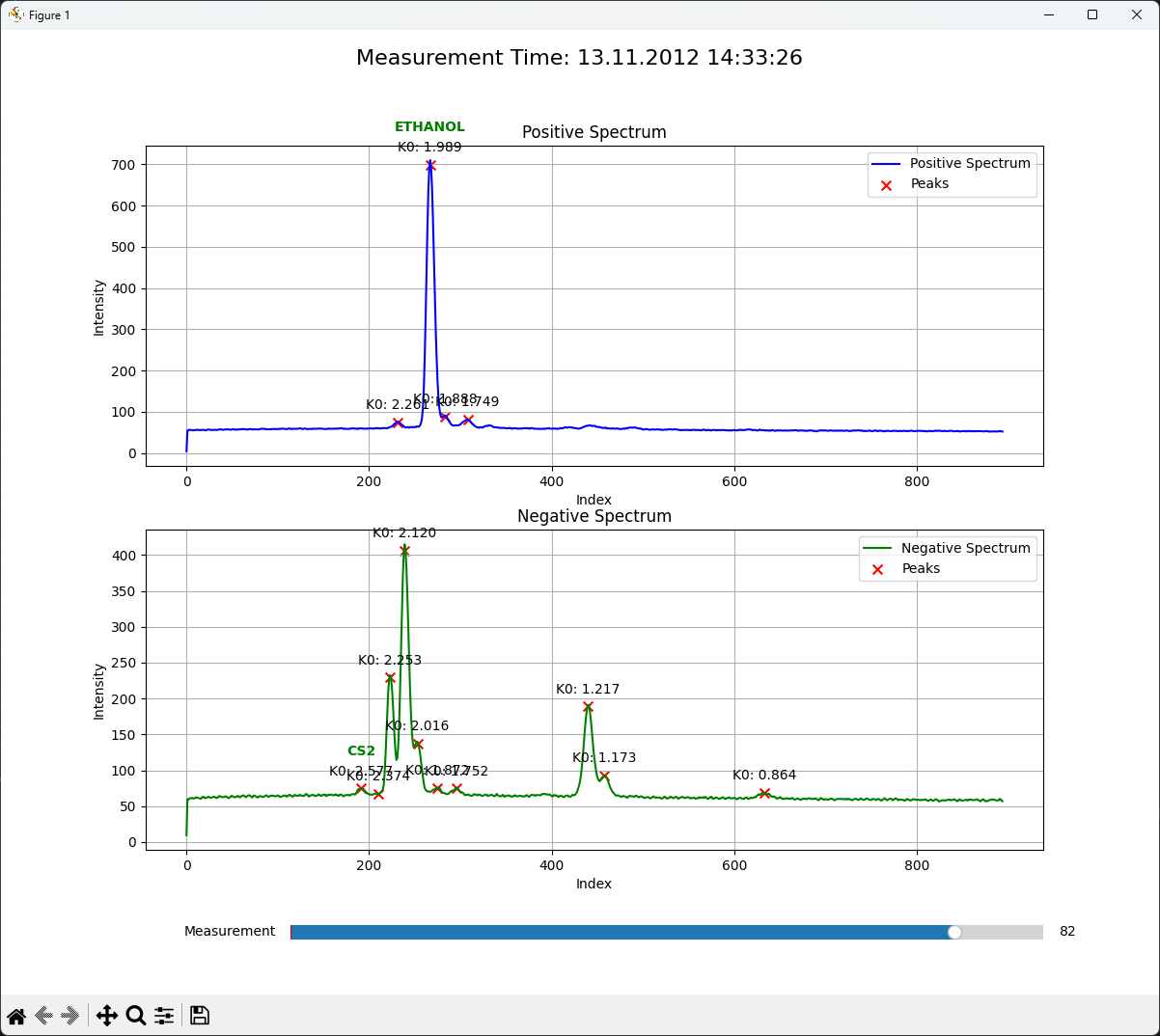


Figure Visualization of the IMS spectra with Ethanol and CS2 substance matching

## 6. Conclusion

In summary, Python proved not only to fulfill all the requirements but also enable rapid, reliable development of a complete IMS data-analysis procedure with a very simple boilerplate. Libraries like *NumPy* and *Pandas* handle large spectral and channel datasets with very good efficiency. *SQLAlchemy* provides a clean, simple and maintainable SQLite interface.  
*MatPlotLib* generate high quality plots with few lines of code.   
Python’s extensive ecosystem facilitates future upgrades and maintainability of the project. Together, these strengths make Python the ideal platform for an end-to-end molecular matching procedure in IMS.

# Ideal Function Identification and Mapping

## 1. Introduction

The goal of this assignment is to make a data analysis workflow based on the Python programming language which identifies mathematical ideal functions that best fit the given training datasets and then maps the test dataset to the ideal functions. The task demonstrates basic principles of data science and software engineering such as object-oriented programming, exception handling, visualization, regression, model evaluation and data mapping.

These components are implemented using existing Python libraries such as *numpy, pandas, sqlalchemy, matplotlib* and some user defined classes and modules.

## 2. Structure of the assignment

### 2.1 Loading and storing data in SQLite

There are three datasets available with which the assignment carries out the identification and mapping procedure: Training data, test data and 50 ideal functions. These datasets are available as CSV files. The data from the CSV files can be inserted into the SQLite database using the *sqlite\_helper.py* module. In the *main.py,* there is a choice given to the user to select how the data should be retrieved. It can be retrieved in three possible ways:

1. Get the ideal, test and train data frames directly from the csv files. This is done by using pandas *read\_csv* method.
2. Read the csv file and insert the data into the respective tables in the database and then retrieve the data from the database as pandas’ data frames. (Not recommended as it may cause duplication of data in the database).
3. Retrieve the data directly from the database (if the data is already in the database).

The *get\_data\_from\_source* function does the work with the default values already set. The user can provide parameters like csv file path, database path and the method of retrieving the data as well.

Custom exception classes such as *TableNotFoundError, ColumnMismatchError and DataInsertionError* ensure robustness by capturing and reporting the errors that occur when some database operations fail.

### 2.2 Identifying the Best Ideal Functions

To identify the ideal functions and to check the best fit of the four training datasets, the *FunctionFitter* class is implemented. It inherits from a base class *BaseFitter* which is already a child class of the *pandas.DataFrame* class. This *FunctionFitter* class uses *Ordinary Least Squares (OLS)* regression by *statsmodels.api.OLS [[6]](#footnote-6)*.

For each ideal function from the dataset, a regression model is fitted using the *FunctionFitter* class against the x-values from the training dataset. Y-values from the prediction are compared with each training dataset / function to calculate the Root Mean Squared Error (RMSE).

The function *choose\_best\_ideal\_functions* calculates and evaluates the RMSE for each ideal-training pair and selects the ideal function which has the least RMSE for each training column. The result obtained is made into a dictionary mapping each training function to its best fitting ideal function along with some extra data like the associated slope, intercept and the calculated RMSE.

Chosen Ideal Functions and RMSE values for the input data set are:

1. y1 -> Ideal Function: y42, RMSE: 0.4676

2. y2 -> Ideal Function: y41, RMSE: 0.4641

3. y3 -> Ideal Function: y11, RMSE: 0.2732

4. y4 -> Ideal Function: y48, RMSE: 0.3266

These results show that ideal functions y42, y41, y11 and y48 best approximate the four training datasets since their RMSE values are the lowest.

### 2.3 Mapping Test Data to Ideal Functions

The next step involved checking whether each x-y pair in the test data can be assigned to any of the selected ideal functions. The *TestMapper* class is responsible for taking test data points and finding out which previously identified ideal function each test point most likely belongs to by calculating which function produces the minimum error when predicting that particular point’s y-value.

*TestMapper* inherits from *BaseMapper* which provides a foundation for mapping functionality. It stores the “best” ideal functions identified earlier and the test data that is to be mapped with the ideal functions. For each test data point, the y-value predicted by the assigned ideal function is compared to the actual y-value. If the absolute deviation is less than or equal to the maximum deviation that we achieve during the training procedure multiplied by , then the test point is a match.

The mapping results are stored in a pandas data frame with the columns: *x, y, assigned\_function* and *deviation.*

Mapping output received for the input set data is printed as:

1. Test Data Mapping Results:

2. x y assigned\_function deviation

3. 0 17.5 34.161040 y41 0.813983

4. 1 0.3 1.215102 y41 0.616719

5. 2 -8.7 -16.843908 y41 0.545485

6. 3 -19.2 -37.170870 y41 1.204263

7. 4 -11.0 -20.263054 y41 1.723215

8. .. ... ... ... ...

9. 95 -1.9 -4.036904 y41 0.238276

10. 96 12.2 -0.010358 y48 0.017739

11. 97 16.5 -33.964134 y42 0.985814

12. 98 5.3 -10.291622 y42 0.285937

13. 99 17.9 28.078455 y41 7.696025

14.

### 2.4 Object-Oriented Programming and Inheritance

The code of the assignment follows object-oriented design principles. The core functionalities are in the classes such as *FunctionFitter, TestMapper* and their base classes. The design helps promote code reusability, clarity and modularity so that in the future, the code can be easily upgraded and maintained, if needed. Inheritance is used to share basic common functions or behavior between parent and child classes such as data loading and mapping mechanism.

### 2.5 Exception Handling

Exception handling is used in the code to catch and report errors in a user-friendly manner. For example, if a specified table is not found in the database, a *TableNotFoundError* is raised. If a CSV’s header does not match the expected table architecture or structure, a *ColumnMismatchError* is thrown. These custom exceptions help improve the maintainability and reliability of the application. The exception.py module can also be upgraded with more user-defined exceptions if needed.

### 2.6 Data Visualization

Data visualization can be done with multiple Python libraries like *Bokeh*, *Matplotlib* and *Seaborn*. This assignment uses *matplotlib.* The module plotdata.py includes the following:

1. *plot\_training\_data*: Visualizes the four training functions.
2. *plot\_ideal\_functions*: Visualizes all the 50 ideal functions.
3. *Plot\_test\_data\_with\_ideal*: Displays test points along with the matched ideal functions.

The plots help verify the correctness of the regression and mapping visually to make sure that the analysis works and it can help the analysis be a bit more transparent.

Example Visualizations:

A graph of lines and curves

AI-generated content may be incorrect.

Figure Ideal Functions Plot: Shows the full range of 50 ideal functions for reference.

A graph with lines and numbers

AI-generated content may be incorrect.

Figure Training Data Plot: Displays noisy linear / nonlinear training data used for fitting.

A graph with lines and dots

AI-generated content may be incorrect.

Figure Test Mapping Plot: Displays test data along with chosen ideal functions.

### 2.7 Unit Testing

To make sure if the stability and correctness of the code is ok, unit tests are written in *test\_all.py*. The *TestFunctionFitter* class tests whether the ideal functions are correctly fitted to the training data and whether the correct functions are selected based on the RMSE calculated. The *TestTestMapper* class is responsible for testing whether the test points are accurately mapped to the ideal functions and whether the deviations calculated are correct.

Some edge cases such as near-boundary devications and incorrect function assigning are covered.

## 3. Conclusion

This assignment shows how Python can be used to effectively handle a data analysis project or problem using regression, model selection etc. Initially, it was a bit challenging to think about how training data to ideal functions properly but breaking the logic into multiple classes helped in managing the complication.

Visualization of the data helped a lot in understanding what mapping of the trained data and the ideal functions too. The most satisfying part was to visualize the mapping of the test data on the selected ideal functions and seeing the plots align. That confirmed that the logic was working.

# Appendix

Both parts of the assignment are published on GitHub as public repositories and standard git workflow has been carried out during the development of the programming part.

“Molecular Matching with Ion Mobility Spectrometry using Python” can be found under <https://github.com/kob824/IMSPeakDetector>

“Ideal Function Identification and Mapping” can be found under <https://github.com/kob824/IdealFunctionAssignment>

## 1. Code

### 1.1 Molecular Matching with Ion Mobility Spectrometry using Python

#### 1.1.1 ims.py

1. import numpy as np

2. from scipy.signal import savgol\_filter, find\_peaks

3. import matplotlib.pyplot as plt

4.

5. def process\_spectrum(spectrum):

6.     spectrum = np.fromstring(spectrum[2:-1], sep=', ')

7.

8.     smoothed\_spectrum = savgol\_filter(spectrum, window\_length=11, polyorder=3)

9.

10.     peaks, properties = find\_peaks(smoothed\_spectrum, height=np.max(smoothed\_spectrum) \* 0.1, distance=5)

11.

12.     top\_peaks = sorted(zip(peaks, properties['peak\_heights']), key=lambda x: x[1], reverse=True)[:10]

13.

14.     return [(int(idx), float(height)) for idx, height in top\_peaks]

15.

16. def calculate\_k0\_value(peaks, temperature, pressure, voltage, drift\_tube\_length, resolution):

17.     """

18.     Calculate the k0 value based on the peaks, temperature, pressure, drift tube length, and resolution.

19.     Returns a list of K0 values for each peak.

20.

21.     The formulas needed are as follows:

22.     drift\_time = (peak\_position + position\_offset) \* (resolution / 1000000)

23.     k = length\_squared / (drift\_time \* voltage)

24.     k0 = k \* (nominal\_temperature / temperature) \* (pressure / nominal\_pressure)

25.     """

26.     nominal\_temperature = 273.15  # Kelvin

27.     nominal\_pressure = 1013.25  # kPa

28.     position\_offset = 31 + 1

29.     length\_squared = drift\_tube\_length \*\* 2

30.     temperature = temperature + 273.15  # Convert to Kelvin

31.

32.     k0\_values = []

33.     for peak in peaks:

34.         peak\_position = peak[0]  # Peak index

35.         drift\_time = (peak\_position + position\_offset) \* (resolution / 1000000)  # Drift time in seconds

36.         k = length\_squared / (drift\_time \* voltage)

37.         k0 = k \* (nominal\_temperature / temperature) \* (pressure / nominal\_pressure)

38.         k0\_values.append(k0)

39.

40.     return k0\_values

41.

#### 1.1.2 substance\_identifier.py

1. import numpy as np

2. from modules import sqlite\_helper

3.

4. # Default tolerance for K0 value matching

5. DEFAULT\_K0\_TOLERANCE = 0.02

6.

7. def identify\_substances(pos\_k0\_values, neg\_k0\_values, tolerance=DEFAULT\_K0\_TOLERANCE):

8.     """

9.     Identify substances by comparing K0 values with the library.

10.     Returns a list of identified substances.

11.     """

12.     library = sqlite\_helper.get\_substance\_library()

13.     identified\_substances = []

14.

15.     for \_, substance in library.iterrows():

16.         substance\_name = substance['substance\_name']

17.

18.         # Check positive spectrum K0 values

19.         pos\_matches = []

20.         for i in range(1, 4):  # Check k0\_pos\_1, k0\_pos\_2, k0\_pos\_3

21.             k0\_col = f'k0\_pos\_{i}'

22.             if substance[k0\_col] > 0:  # Only check non-zero values

23.                 # Check if this K0 value matches any in our detected peaks

24.                 for k0 in pos\_k0\_values:

25.                     if abs(k0 - substance[k0\_col]) <= tolerance:

26.                         pos\_matches.append((k0\_col, k0, substance[k0\_col]))

27.                         break

28.

29.         # Check negative spectrum K0 values

30.         neg\_matches = []

31.         for i in range(1, 4):  # Check k0\_neg\_1, k0\_neg\_2, k0\_neg\_3

32.             k0\_col = f'k0\_neg\_{i}'

33.             if substance[k0\_col] > 0:  # Only check non-zero values

34.                 # Check if this K0 value matches any in our detected peaks

35.                 for k0 in neg\_k0\_values:

36.                     if abs(k0 - substance[k0\_col]) <= tolerance:

37.                         neg\_matches.append((k0\_col, k0, substance[k0\_col]))

38.                         break

39.

40.         # Count required matches (non-zero K0 values in library)

41.         required\_pos\_matches = sum(1 for i in range(1, 4) if substance[f'k0\_pos\_{i}'] > 0)

42.         required\_neg\_matches = sum(1 for i in range(1, 4) if substance[f'k0\_neg\_{i}'] > 0)

43.

44.         # Substance is identified if all required K0 values match

45.         if (len(pos\_matches) == required\_pos\_matches and required\_pos\_matches > 0) or \

46.            (len(neg\_matches) == required\_neg\_matches and required\_neg\_matches > 0):

47.             identified\_substances.append({

48.                 'name': substance\_name,

49.                 'pos\_matches': pos\_matches,

50.                 'neg\_matches': neg\_matches

51.             })

52.

53.     return identified\_substances

54.

#### 1.1.3 sqlite\_helper.py

1. import pandas as pd

2. from sqlalchemy import create\_engine, Column, Integer, String, Float, BLOB, Table, MetaData

3. from sqlalchemy.ext.declarative import declarative\_base

4. from sqlalchemy.orm import sessionmaker

5. import os

6.

7. DB\_FILE = './db/ims.db'

8. engine = create\_engine(f'sqlite:///{DB\_FILE}')

9. Base = declarative\_base()

10.

11. class Measurement(Base):

12.     \_\_tablename\_\_ = 'measurements'

13.

14.     id = Column(Integer, primary\_key=True, autoincrement=True)

15.     measurement\_time = Column(String)

16.     channel\_1 = Column(Float)

17.     channel\_2 = Column(Float)

18.     channel\_3 = Column(Float)

19.     channel\_4 = Column(Float)

20.     channel\_5 = Column(Float)

21.     channel\_6 = Column(Float)

22.     channel\_7 = Column(Float)

23.     channel\_8 = Column(Float)

24.     dilution = Column(Float)

25.     temperature\_drift\_tube = Column(Float)

26.     pressure = Column(Float)

27.     pos\_voltage = Column(Float)

28.     neg\_voltage = Column(Float)

29.     tube\_length = Column(Float)

30.     pressure\_offset = Column(Float)

31.     pressure\_gradient = Column(Float)

32.     pos\_spectrum = Column(BLOB)

33.     neg\_spectrum = Column(BLOB)

34.

35. Base.metadata.create\_all(engine)

36.

37. Session = sessionmaker(bind=engine)

38. session = Session()

39.

40. def select\_columns\_from\_db(columns, table='measurements'):

41.     """

42.     Select columns from the specified table.

43.

44.     Parameters:

45.     - columns: List of column names

46.     - table: Table name (default: 'measurements')

47.

48.     Returns:

49.     - DataFrame with selected data

50.     """

51.     if table == 'measurements':

52.         query = session.query(\*[getattr(Measurement, col) for col in columns])

53.         results = query.all()

54.         df = pd.DataFrame(results, columns=columns)

55.     else:

56.         # For other tables, use raw SQL

57.         columns\_str = ', '.join(columns)

58.         query = f"SELECT {columns\_str} FROM {table}"

59.         df = pd.read\_sql\_query(query, engine)

60.

61.     return df

62.

63. def load\_csv\_and\_insert(csv\_file):

64.     df = pd.read\_csv(csv\_file, header=0)

65.     print("DataFrame head:")

66.     print(df.head())

67.

68.     for index, row in df.iterrows():

69.         try:

70.             measurement = Measurement(

71.                 measurement\_time = row["measurement\_time"],

72.                 channel\_1 = float(row["channel\_1"]),

73.                 channel\_2 = float(row["channel\_2"]),

74.                 channel\_3 = float(row["channel\_3"]),

75.                 channel\_4 = float(row["channel\_4"]),

76.                 channel\_5 = float(row["channel\_5"]),

77.                 channel\_6 = float(row["channel\_6"]),

78.                 channel\_7 = float(row["channel\_7"]),

79.                 channel\_8 = float(row["channel\_8"]),

80.                 dilution = float(row["dilution"]),

81.                 temperature\_drift\_tube = float(row["temperature\_drift\_tube"]),

82.                 pressure = float(row["pressure"]),

83.                 pos\_voltage = float(row["pos\_voltage"]),

84.                 neg\_voltage = float(row["neg\_voltage"]),

85.                 tube\_length = float(row["tube\_length"]),

86.                 pressure\_offset = float(row["press\_offset"]),

87.                 pressure\_gradient = float(row["press\_gradient"]),

88.                 pos\_spectrum = row["pos\_spectrum"].encode('utf-8'),

89.                 neg\_spectrum = row["neg\_spectrum"].encode('utf-8')

90.             )

91.             session.add(measurement)

92.         except ValueError as ve:

93.             print(f"Error converting row {index}: {ve}")

94.

95.     session.commit()

96.     print(f"Inserted {len(df)} records into the database.")

97.

98. def get\_substance\_library():

99.     """

100.     Retrieve the substance library with K0 values from the database.

101.     """

102.     columns = ["id", "substance\_name",

103.                "k0\_pos\_1", "k0\_pos\_2", "k0\_pos\_3",

104.                "k0\_neg\_1", "k0\_neg\_2", "k0\_neg\_3"]

105.

106.     return select\_columns\_from\_db(columns, table='library')

107.

#### 1.1.4 visualization.py

1. import matplotlib.pyplot as plt

2. import numpy as np

3. from matplotlib.widgets import Slider

4.

5. # Constants for annotation and visualization

6. K0\_ANNOTATION\_OFFSET = (0, 10)

7. SUBSTANCE\_ANNOTATION\_OFFSET = (0, 25)

8. K0\_MATCH\_TOLERANCE = 0.02

9. PEAK\_MARKER\_SIZE = 50

10.

11. def create\_spectrum\_plot(spectrums, peaks\_data, identified\_substances=None):

12.     """

13.     Create a plot showing spectra with peaks and identified substances.

14.

15.     Parameters:

16.     - spectrums: dictionary with 'pos' and 'neg' arrays

17.     - peaks\_data: dictionary with 'pos' and 'neg' peak information (index, height)

18.     - identified\_substances: list of identified substances with their K0 matches

19.     """

20.     fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

21.

22.     # Plot positive spectrum

23.     if 'pos' in spectrums and spectrums['pos'] is not None and len(spectrums['pos']) > 0:

24.         ax1.plot(spectrums['pos'], color='blue', label='Positive Spectrum')

25.

26.         if 'pos' in peaks\_data and peaks\_data['pos']:

27.             peak\_indices = [p[0] for p in peaks\_data['pos']]

28.             peak\_heights = [p[1] for p in peaks\_data['pos']]

29.             ax1.scatter(peak\_indices, peak\_heights, color='red', s=PEAK\_MARKER\_SIZE, marker='x', label='Peaks')

30.

31.             # Annotate peaks with K0 values

32.             for i, (idx, height) in enumerate(zip(peak\_indices, peak\_heights)):

33.                 ax1.annotate(f"K0: {peaks\_data['pos\_k0s'][i]:.3f}",

34.                             (idx, height),

35.                             textcoords="offset points",

36.                             xytext=K0\_ANNOTATION\_OFFSET,

37.                             ha='center')

38.

39.                 # Mark identified substances

40.                 if identified\_substances:

41.                     for substance in identified\_substances:

42.                         for match in substance['pos\_matches']:

43.                             if abs(peaks\_data['pos\_k0s'][i] - match[2]) <= K0\_MATCH\_TOLERANCE:

44.                                 ax1.annotate(f"{substance['name']}",

45.                                            (idx, height),

46.                                            textcoords="offset points",

47.                                            xytext=SUBSTANCE\_ANNOTATION\_OFFSET,

48.                                            ha='center',

49.                                            color='green',

50.                                            weight='bold')

51.

52.     # Plot negative spectrum

53.     if 'neg' in spectrums and spectrums['neg'] is not None and len(spectrums['neg']) > 0:

54.         ax2.plot(spectrums['neg'], color='green', label='Negative Spectrum')

55.

56.         if 'neg' in peaks\_data and peaks\_data['neg']:

57.             peak\_indices = [p[0] for p in peaks\_data['neg']]

58.             peak\_heights = [p[1] for p in peaks\_data['neg']]

59.             ax2.scatter(peak\_indices, peak\_heights, color='red', s=PEAK\_MARKER\_SIZE, marker='x', label='Peaks')

60.

61.             # Annotate peaks with K0 values

62.             for i, (idx, height) in enumerate(zip(peak\_indices, peak\_heights)):

63.                 ax2.annotate(f"K0: {peaks\_data['neg\_k0s'][i]:.3f}",

64.                             (idx, height),

65.                             textcoords="offset points",

66.                             xytext=K0\_ANNOTATION\_OFFSET,

67.                             ha='center')

68.

69.                 # Mark identified substances

70.                 if identified\_substances:

71.                     for substance in identified\_substances:

72.                         for match in substance['neg\_matches']:

73.                             if abs(peaks\_data['neg\_k0s'][i] - match[2]) <= K0\_MATCH\_TOLERANCE:

74.                                 ax2.annotate(f"{substance['name']}",

75.                                            (idx, height),

76.                                            textcoords="offset points",

77.                                            xytext=SUBSTANCE\_ANNOTATION\_OFFSET,

78.                                            ha='center',

79.                                            color='green',

80.                                            weight='bold')

81.

82.     ax1.set\_title('Positive Spectrum')

83.     ax1.set\_xlabel('Index')

84.     ax1.set\_ylabel('Intensity')

85.     ax1.legend()

86.     ax1.grid(True)

87.

88.     ax2.set\_title('Negative Spectrum')

89.     ax2.set\_xlabel('Index')

90.     ax2.set\_ylabel('Intensity')

91.     ax2.legend()

92.     ax2.grid(True)

93.

94.     plt.tight\_layout()

95.     return fig

96.

97. def show\_scrollable\_plots(data\_list):

98.     """

99.     Create a scrollable interface to navigate through multiple spectrum plots.

100.

101.     Parameters:

102.     - data\_list: List of dictionaries containing spectrum data for each measurement

103.     """

104.     if not data\_list:

105.         print("No data to display")

106.         return

107.

108.     # Create a figure that will persist

109.     fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(12, 10))

110.     # Slider position adjustment

111.     SLIDER\_BOTTOM\_MARGIN = 0.15

112.     plt.subplots\_adjust(bottom=SLIDER\_BOTTOM\_MARGIN)

113.

114.     # Function to update the plot

115.     def update\_plot(idx):

116.         ax1.clear()

117.         ax2.clear()

118.

119.         current\_data = data\_list[idx]

120.

121.         # Plot positive spectrum

122.         if current\_data['spectrums']['pos'] is not None:

123.             pos\_spectrum = current\_data['spectrums']['pos']

124.             ax1.plot(pos\_spectrum, color='blue', label='Positive Spectrum')

125.

126.             if current\_data['peaks\_data']['pos']:

127.                 peak\_indices = [p[0] for p in current\_data['peaks\_data']['pos']]

128.                 peak\_heights = [p[1] for p in current\_data['peaks\_data']['pos']]

129.                 ax1.scatter(peak\_indices, peak\_heights, color='red', s=PEAK\_MARKER\_SIZE, marker='x', label='Peaks')

130.

131.                 # Annotate peaks with K0 values

132.                 for i, (idx, height) in enumerate(zip(peak\_indices, peak\_heights)):

133.                     ax1.annotate(f"K0: {current\_data['peaks\_data']['pos\_k0s'][i]:.3f}",

134.                                 (idx, height),

135.                                 textcoords="offset points",

136.                                 xytext=K0\_ANNOTATION\_OFFSET,

137.                                 ha='center')

138.

139.                     # Mark identified substances

140.                     if current\_data['identified\_substances']:

141.                         for substance in current\_data['identified\_substances']:

142.                             for match in substance['pos\_matches']:

143.                                 if abs(current\_data['peaks\_data']['pos\_k0s'][i] - match[2]) <= K0\_MATCH\_TOLERANCE:

144.                                     ax1.annotate(f"{substance['name']}",

145.                                                (idx, height),

146.                                                textcoords="offset points",

147.                                                xytext=SUBSTANCE\_ANNOTATION\_OFFSET,

148.                                                ha='center',

149.                                                color='green',

150.                                                weight='bold')

151.

152.         # Plot negative spectrum

153.         if current\_data['spectrums']['neg'] is not None:

154.             neg\_spectrum = current\_data['spectrums']['neg']

155.             ax2.plot(neg\_spectrum, color='green', label='Negative Spectrum')

156.

157.             if current\_data['peaks\_data']['neg']:

158.                 peak\_indices = [p[0] for p in current\_data['peaks\_data']['neg']]

159.                 peak\_heights = [p[1] for p in current\_data['peaks\_data']['neg']]

160.                 ax2.scatter(peak\_indices, peak\_heights, color='red', s=PEAK\_MARKER\_SIZE, marker='x', label='Peaks')

161.

162.                 # Annotate peaks with K0 values

163.                 for i, (idx, height) in enumerate(zip(peak\_indices, peak\_heights)):

164.                     ax2.annotate(f"K0: {current\_data['peaks\_data']['neg\_k0s'][i]:.3f}",

165.                                 (idx, height),

166.                                 textcoords="offset points",

167.                                 xytext=K0\_ANNOTATION\_OFFSET,

168.                                 ha='center')

169.

170.                     # Mark identified substances

171.                     if current\_data['identified\_substances']:

172.                         for substance in current\_data['identified\_substances']:

173.                             for match in substance['neg\_matches']:

174.                                 if abs(current\_data['peaks\_data']['neg\_k0s'][i] - match[2]) <= K0\_MATCH\_TOLERANCE:

175.                                     ax2.annotate(f"{substance['name']}",

176.                                                (idx, height),

177.                                                textcoords="offset points",

178.                                                xytext=SUBSTANCE\_ANNOTATION\_OFFSET,

179.                                                ha='center',

180.                                                color='green',

181.                                                weight='bold')

182.

183.         ax1.set\_title('Positive Spectrum')

184.         ax1.set\_xlabel('Index')

185.         ax1.set\_ylabel('Intensity')

186.         ax1.legend()

187.         ax1.grid(True)

188.

189.         ax2.set\_title('Negative Spectrum')

190.         ax2.set\_xlabel('Index')

191.         ax2.set\_ylabel('Intensity')

192.         ax2.legend()

193.         ax2.grid(True)

194.

195.         fig.suptitle(f"Measurement Time: {current\_data['measurement\_time']}", fontsize=16)

196.         fig.canvas.draw\_idle()

197.

198.     # Initial plot

199.     update\_plot(0)

200.

201.     # Add slider

202.     SLIDER\_X\_POSITION = 0.25

203.     SLIDER\_Y\_POSITION = 0.05

204.     SLIDER\_WIDTH = 0.65

205.     SLIDER\_HEIGHT = 0.03

206.     ax\_slider = plt.axes([SLIDER\_X\_POSITION, SLIDER\_Y\_POSITION, SLIDER\_WIDTH, SLIDER\_HEIGHT])

207.     slider = Slider(

208.         ax=ax\_slider,

209.         label='Measurement',

210.         valmin=0,

211.         valmax=len(data\_list) - 1,

212.         valinit=0,

213.         valstep=1

214.     )

215.

216.     # Connect the slider to the update function

217.     def update(val):

218.         update\_plot(int(slider.val))

219.

220.     slider.on\_changed(update)

221.     plt.show()

222.

223.

#### 1.1.5 main.py

1. from modules import sqlite\_helper

2. from modules.ims import process\_spectrum, calculate\_k0\_value

3. from modules.substance\_identifier import identify\_substances, DEFAULT\_K0\_TOLERANCE

4. from modules.visualization import show\_scrollable\_plots

5. import numpy as np

6. import pandas as pd

7.

8. # Constants

9. K0\_TOLERANCE = 0.2  # The tolerance used specifically in this script

10.

11. def main():

12.     # csv\_file = r"./data/HCl 10ppm.csv"

13.     # sqlite\_helper.load\_csv\_and\_insert(csv\_file)

14.

15.     df = sqlite\_helper.select\_columns\_from\_db(["measurement\_time", "pos\_spectrum", "neg\_spectrum",

16.                                               "temperature\_drift\_tube", "pressure", "pos\_voltage",

17.                                               "neg\_voltage", "tube\_length"])

18.

19.     # Most of the rows in the database have either positive or negative spectrum data, not both

20.     # We need to merge them based on the presence of data in either spectrum

21.     # Process and merge the data to combine appropriate positive and negative spectra

22.     merged\_data = []

23.     i = 0

24.     while i < len(df):

25.         current\_row = df.iloc[i].copy()

26.         pos\_spectrum = current\_row["pos\_spectrum"]

27.         neg\_spectrum = current\_row["neg\_spectrum"]

28.

29.         pos\_spectrum\_array = np.fromstring(pos\_spectrum[2:-1], sep=', ')

30.         neg\_spectrum\_array = np.fromstring(neg\_spectrum[2:-1], sep=', ')

31.

32.         # Case 1: Both spectra have data - no need to merge

33.         if not np.all(pos\_spectrum\_array == 0) and not np.all(neg\_spectrum\_array == 0):

34.             merged\_data.append(current\_row)

35.             i += 1

36.             continue

37.

38.         # Case 2: Current row has positive spectrum but no negative spectrum

39.         if not np.all(pos\_spectrum\_array == 0) and np.all(neg\_spectrum\_array == 0):

40.             # Look ahead for a row with negative spectrum data

41.             next\_neg\_idx = -1

42.             for j in range(i+1, len(df)):

43.                 next\_neg\_array = np.fromstring(df.iloc[j]["neg\_spectrum"][2:-1], sep=', ')

44.                 next\_pos\_array = np.fromstring(df.iloc[j]["pos\_spectrum"][2:-1], sep=', ')

45.

46.                 # If we find a row with negative data and no positive data

47.                 if not np.all(next\_neg\_array == 0) and np.all(next\_pos\_array == 0):

48.                     next\_neg\_idx = j

49.                     break

50.

51.             if next\_neg\_idx != -1:

52.                 # Merge the data - keep metadata from current row with positive spectrum

53.                 merged\_row = current\_row.copy()

54.                 merged\_row["neg\_spectrum"] = df.iloc[next\_neg\_idx]["neg\_spectrum"]

55.                 merged\_data.append(merged\_row)

56.                 i = next\_neg\_idx + 1  # Skip to after the last used row

57.             else:

58.                 # No matching negative spectrum found

59.                 i += 1

60.             continue

61.

62.         # Case 3: Current row has negative spectrum but no positive spectrum

63.         if np.all(pos\_spectrum\_array == 0) and not np.all(neg\_spectrum\_array == 0):

64.             # Look behind for the most recent row with positive spectrum data

65.             prev\_pos\_idx = -1

66.             for j in range(i-1, -1, -1):

67.                 prev\_pos\_array = np.fromstring(df.iloc[j]["pos\_spectrum"][2:-1], sep=', ')

68.

69.                 # If we find a row with positive data

70.                 if not np.all(prev\_pos\_array == 0):

71.                     prev\_pos\_idx = j

72.                     break

73.

74.             if prev\_pos\_idx != -1:

75.                 # We've already processed this by looking ahead from the positive spectrum row

76.                 i += 1

77.             else:

78.                 # No matching positive spectrum found before this

79.                 i += 1

80.             continue

81.

82.         # Case 4: Both spectra are empty

83.         i += 1

84.

85.     visualization\_data = []

86.

87.     for i, row in enumerate(merged\_data):

88.         measurement\_time = row["measurement\_time"]

89.         pos\_spectrum = row["pos\_spectrum"]

90.         neg\_spectrum = row["neg\_spectrum"]

91.         temperature = row["temperature\_drift\_tube"]

92.         pressure = row["pressure"]

93.         pos\_voltage = row["pos\_voltage"]

94.         neg\_voltage = row["neg\_voltage"]

95.         drift\_tube\_length = row["tube\_length"]

96.         resolution = 32.392

97.

98.         pos\_spectrum\_array = np.fromstring(pos\_spectrum[2:-1], sep=', ')

99.         neg\_spectrum\_array = np.fromstring(neg\_spectrum[2:-1], sep=', ')

100.

101.         # Skip if either spectrum is empty

102.         if np.all(pos\_spectrum\_array == 0) or np.all(neg\_spectrum\_array == 0):

103.             print(f"Skipping spectrum at Measurement Time: {measurement\_time} (contains zeros in one spectrum)")

104.             continue

105.

106.         print(f"\nProcessing Measurement Time: {measurement\_time} (Entry {i+1}/{len(merged\_data)})")

107.

108.         pos\_top\_peaks = []

109.         pos\_k0\_values = []

110.         if not np.all(pos\_spectrum\_array == 0):

111.             pos\_top\_peaks = process\_spectrum(pos\_spectrum)

112.             pos\_k0\_values = calculate\_k0\_value(pos\_top\_peaks, temperature, pressure, pos\_voltage, drift\_tube\_length, resolution)

113.             print(f"Positive Spectrum:")

114.             print(f"Top 10 Peaks: {pos\_top\_peaks}")

115.             print(f"K0 Values: {pos\_k0\_values}")

116.

117.         neg\_top\_peaks = []

118.         neg\_k0\_values = []

119.         if not np.all(neg\_spectrum\_array == 0):

120.             neg\_top\_peaks = process\_spectrum(neg\_spectrum)

121.             neg\_k0\_values = calculate\_k0\_value(neg\_top\_peaks, temperature, pressure, neg\_voltage, drift\_tube\_length, resolution)

122.             print(f"Negative Spectrum:")

123.             print(f"Top 10 Peaks: {neg\_top\_peaks}")

124.             print(f"K0 Values: {neg\_k0\_values}")

125.

126.         # Identify substances based on K0 values

127.         # identified\_substances = identify\_substances(pos\_k0\_values, neg\_k0\_values, tolerance=0.02)

128.         identified\_substances = identify\_substances(pos\_k0\_values, neg\_k0\_values, tolerance=K0\_TOLERANCE)

129.

130.         if identified\_substances:

131.             print(f"\nIdentified Substances:")

132.             for substance in identified\_substances:

133.                 print(f"- {substance['name']}")

134.                 if substance['pos\_matches']:

135.                     print(f"  Positive spectrum matches:")

136.                     for match in substance['pos\_matches']:

137.                         print(f"    {match[0]} library value: {match[2]:.3f}, measured: {match[1]:.3f}")

138.                 if substance['neg\_matches']:

139.                     print(f"  Negative spectrum matches:")

140.                     for match in substance['neg\_matches']:

141.                         print(f"    {match[0]} library value: {match[2]:.3f}, measured: {match[1]:.3f}")

142.         else:

143.             print("\nNo substances identified in this spectrum.")

144.

145.         print("="\*50)

146.

147.         # data for visualization

148.         visualization\_data.append({

149.             'measurement\_time': measurement\_time,

150.             'spectrums': {

151.                 'pos': pos\_spectrum\_array,

152.                 'neg': neg\_spectrum\_array

153.             },

154.             'peaks\_data': {

155.                 'pos': pos\_top\_peaks,

156.                 'neg': neg\_top\_peaks,

157.                 'pos\_k0s': pos\_k0\_values,

158.                 'neg\_k0s': neg\_k0\_values

159.             },

160.             'identified\_substances': identified\_substances

161.         })

162.

163.     print(f"\nFound {len(visualization\_data)} complete spectra (with both positive and negative data)")

164.     print (f"\nTotal matches found: {sum(len(data['identified\_substances']) for data in visualization\_data)}")

165.

166.     show\_scrollable\_plots(visualization\_data)

167.

168. if \_\_name\_\_ == "\_\_main\_\_":

169.     main()

170.

### 1.2 Ideal Function Identification and Mapping

#### 1.2.1 idealfunctionsmodule.py

1. import pandas as pd

2. import numpy as np

3. import statsmodels.api as sm

4. from sklearn.metrics import mean\_squared\_error

5. import math

6.

7. class BaseFitter(pd.DataFrame):  # Inheriting from pandas DataFrame class

8.     """

9.     Base class for fitting functions to data. Inherits from pandas DataFrame.

10.

11.     Attributes:

12.         ideal\_df (pd.DataFrame): DataFrame containing ideal functions.

13.         train\_df (pd.DataFrame): DataFrame containing training data.

14.     """

15.     def \_\_init\_\_(self, ideal\_df, train\_df):

16.         """

17.         Initialize the BaseFitter with ideal and training data.

18.

19.         Args:

20.             ideal\_df (pd.DataFrame): DataFrame containing ideal functions.

21.             train\_df (pd.DataFrame): DataFrame containing training data.

22.         """

23.         super().\_\_init\_\_()  # Initialize the parent class

24.         self.ideal\_df = ideal\_df

25.         self.train\_df = train\_df

26.

27. class FunctionFitter(BaseFitter):

28.     """

29.     Class for fitting ideal functions to training data and selecting the best functions.

30.

31.     Attributes:

32.         ideal\_models (dict): Dictionary to store fitted models and their parameters.

33.         rmse\_results (dict): Dictionary to store RMSE results for each model.

34.     """

35.     def \_\_init\_\_(self, ideal\_df, train\_df):

36.         """

37.         Initialize the FunctionFitter with ideal and training data.

38.

39.         Args:

40.             ideal\_df (pd.DataFrame): DataFrame containing ideal functions.

41.             train\_df (pd.DataFrame): DataFrame containing training data.

42.         """

43.         super().\_\_init\_\_(ideal\_df, train\_df)  # Call the parent class constructor

44.         self.ideal\_models = {}

45.         self.rmse\_results = {}  # Dictionary to store RMSE results for each model

46.

47.     def fit\_ideal\_functions(self):

48.         """

49.         Fit ideal functions to the training data and calculate RMSE for each combination.

50.         """

51.         train\_x = self.train\_df['x']

52.         train\_x\_const = sm.add\_constant(train\_x)  # Add a constant term to the predictor

53.

54.         for col in self.ideal\_df.columns:

55.             if col == 'x':

56.                 continue  # Skip the x column

57.

58.             x\_values = self.ideal\_df['x']

59.             y\_values = self.ideal\_df[col]

60.             x\_const = sm.add\_constant(x\_values)

61.             model = sm.OLS(y\_values, x\_const).fit()  # Fit the model

62.

63.             # Calculate the slope and intercept

64.             slope = model.params['x']

65.             intercept = model.params['const']

66.

67.             # Prediction on the training x values using the ideal function model

68.             train\_prediction = model.predict(train\_x\_const)

69.

70.             # Calculate the Root Mean Squared Error using the training data

71.             rmse\_dict = {}

72.             for train\_col in self.train\_df.columns:

73.                 if train\_col == 'x':

74.                     continue  # Skip the x column

75.                 y\_true = self.train\_df[train\_col]

76.                 rmse = math.sqrt(mean\_squared\_error(y\_true, train\_prediction))

77.                 rmse\_dict[train\_col] = rmse

78.

79.             # Save model values and RMSE values

80.             self.ideal\_models[col] = {'model': model, 'slope': slope, 'intercept': intercept}

81.             self.rmse\_results[col] = rmse\_dict

82.

83.     def choose\_best\_ideal\_functions(self):

84.         """

85.         Select the best ideal function for each training column based on RMSE.

86.

87.         Returns:

88.             dict: Dictionary containing the best ideal function, slope, intercept, and RMSE for each training column.

89.         """

90.         best\_functions = {}

91.         training\_columns = [col for col in self.train\_df.columns if col != 'x']  # Skip the x column

92.         for train\_col in training\_columns:

93.             best\_rmse = float('inf')

94.             best\_ideal = None

95.             for ideal\_func, rmse\_dict in self.rmse\_results.items():

96.                 if rmse\_dict[train\_col] < best\_rmse:

97.                     best\_rmse = rmse\_dict[train\_col]

98.                     best\_ideal = ideal\_func

99.             best\_functions[train\_col] = {

100.                 'ideal\_function': best\_ideal,

101.                 'slope': self.ideal\_models[best\_ideal]['slope'],

102.                 'intercept': self.ideal\_models[best\_ideal]['intercept'],

103.                 'rmse': best\_rmse

104.             }

105.         return best\_functions

106.

107. class BaseMapper:

108.     """

109.     Base class for mapping test data to the best ideal functions.

110.

111.     Attributes:

112.         best\_functions (dict): Dictionary of best ideal functions and their parameters.

113.         test\_df (pd.DataFrame): DataFrame containing test data.

114.         mapping\_results (pd.DataFrame): DataFrame to store mapping results.

115.     """

116.     def \_\_init\_\_(self, best\_functions, test\_df):

117.         """

118.         Initialize the BaseMapper with best functions and test data.

119.

120.         Args:

121.             best\_functions (dict): Dictionary of best ideal functions and their parameters.

122.             test\_df (pd.DataFrame): DataFrame containing test data.

123.         """

124.         self.best\_functions = best\_functions

125.         self.test\_df = test\_df

126.         self.mapping\_results = pd.DataFrame(columns=['x', 'y', 'assigned\_function', 'deviation'])

127.

128.     def map\_test\_data(self):

129.         """

130.         Abstract method to map test data to ideal functions. Must be implemented by subclasses.

131.         """

132.         raise NotImplementedError("This method should be implemented by subclasses.")

133.

134. class TestMapper(BaseMapper):

135.     """

136.     Class for mapping test data to the best ideal functions.

137.

138.     Attributes:

139.         Inherits all attributes from BaseMapper.

140.     """

141.     def \_\_init\_\_(self, best\_functions, test\_df):

142.         """

143.         Initialize the TestMapper with best functions and test data.

144.

145.         Args:

146.             best\_functions (dict): Dictionary of best ideal functions and their parameters.

147.             test\_df (pd.DataFrame): DataFrame containing test data.

148.         """

149.         super().\_\_init\_\_(best\_functions, test\_df)  # Call the parent class constructor

150.

151.     def map\_test\_data(self):

152.         """

153.         Map test data points to the best ideal functions based on minimum deviation.

154.

155.         Returns:

156.             pd.DataFrame: DataFrame containing mapping results with columns ['x', 'y', 'assigned\_function', 'deviation'].

157.         """

158.         results = []

159.         for \_, row in self.test\_df.iterrows():

160.             x\_val = row['x']

161.             y\_actual = row['y']

162.             best\_error = float('inf')

163.             assigned\_function = None

164.             # Loop through the best ideal functions.

165.             for train\_col, func\_info in self.best\_functions.items():

166.                 slope = func\_info['slope']

167.                 intercept = func\_info['intercept']

168.

169.                 # Calculate the predicted y value using the ideal model

170.                 y\_pred = slope \* x\_val + intercept

171.                 error = abs(y\_actual - y\_pred)

172.                 if error < best\_error:

173.                     best\_error = error

174.                     assigned\_function = func\_info['ideal\_function']

175.             results.append({

176.                 'x': x\_val,

177.                 'y': y\_actual,

178.                 'assigned\_function': assigned\_function,

179.                 'deviation': best\_error

180.             })

181.         self.mapping\_results = pd.DataFrame(results)

182.         return self.mapping\_results

183.

#### 1.2.2 sqlite\_helper.py

1. from sqlalchemy import create\_engine, inspect

2. import pandas as pd

3.

4. try:

5.     # When running from the main module (e.g., main.py)

6.     from modules import exceptions as ex

7. except ImportError:

8.     # When running directly from this module

9.     import exceptions as ex

10.

11. def insert\_csv\_to\_table(csv\_file\_path, db\_path):

12.     """

13.     Inserts data from a CSV file into a SQLite table if the table exists.

14.     The table name is derived from the CSV file name (without extension).

15.     Headers in the CSV file must match the table columns.

16.

17.     Args:

18.         csv\_file\_path (str): Path to the CSV file.

19.         db\_path (str): Path to the SQLite database.

20.     """

21.     table\_name = csv\_file\_path.split('/')[-1].split('.')[0]

22.

23.     engine = create\_engine(f'sqlite:///{db\_path}')

24.

25.     inspector = inspect(engine)

26.     if table\_name not in inspector.get\_table\_names():

27.         raise ex.TableNotFoundError(table\_name)

28.

29.     df = pd.read\_csv(csv\_file\_path)

30.

31.     try:

32.         df.to\_sql(table\_name, engine, if\_exists='append', index=False)

33.         print(f"Data from '{csv\_file\_path}' inserted into table '{table\_name}' successfully.")

34.     except Exception as e:

35.         raise ex.DataInsertionError(table\_name, str(e))

36.

37. def get\_table\_data\_as\_df(db\_path, table\_name):

38.     """

39.     Retrieves data from a specified SQLite table and returns it as a pandas DataFrame,

40.     excluding the 'id' column if it exists.

41.

42.     Args:

43.         db\_path (str): Path to the SQLite database.

44.         table\_name (str): Name of the table to retrieve data from.

45.

46.     Returns:

47.         pd.DataFrame: DataFrame containing the data from the specified table, excluding the 'id' column.

48.     """

49.     # Create a database engine

50.     engine = create\_engine(f'sqlite:///{db\_path}')

51.

52.     # Load data from the specified table into a DataFrame

53.     df = pd.read\_sql\_table(table\_name, engine)

54.

55.     # Drop the 'id' column if it exists

56.     if 'id' in df.columns:

57.         df = df.drop(columns=['id'])

58.

59.     return df

60.

61. # Small test block to check if the exceptions work

62.

63. # if \_\_name\_\_ == "\_\_main\_\_":

64. #     try:

65. #         insert\_csv\_to\_table("non\_existent.csv", "test.db")

66. #     except ex.TableNotFoundError as e:

67. #         print(e)

68.

69. #     try:

70. #         insert\_csv\_to\_table("data/ideal.csv", "test.db")

71. #     except ex.DataInsertionError as e:

72. #         print(e)

73.

#### 1.2.3 plotdata.py

1. import matplotlib.pyplot as plt

2. import numpy as np

3.

4. def plot\_training\_data(train\_df):

5.     """

6.     Plots the training data.

7.     """

8.     plt.figure(figsize=(10, 6))

9.     for col in train\_df.columns:

10.         if col == 'x':

11.             continue

12.         plt.plot(train\_df['x'], train\_df[col], linestyle='--', label=f"Train {col}")

13.     plt.xlabel("x")

14.     plt.ylabel("y")

15.     plt.title("Training Data")

16.     plt.legend()

17.     plt.grid(True)

18.     plt.show()

19.

20. def plot\_ideal\_functions(ideal\_df):

21.     """

22.     Plots all ideal functions.

23.     """

24.     plt.figure(figsize=(10, 6))

25.     for col in ideal\_df.columns:

26.         if col == 'x':

27.             continue

28.         plt.plot(ideal\_df['x'], ideal\_df[col], label=f"Ideal {col}")

29.     plt.xlabel("x")

30.     plt.ylabel("y")

31.     plt.title("Ideal Functions")

32.     plt.legend()

33.     plt.grid(True)

34.     plt.show()

35.

36. def plot\_test\_data\_with\_ideal(test\_df, ideal\_df, best\_functions):

37.     """

38.     Plots the test data along with the chosen ideal functions.

39.     """

40.     plt.figure(figsize=(10, 6))

41.     colors = ['r', 'g', 'b', 'm']

42.     x\_vals = ideal\_df['x']

43.     i = 0

44.     for train\_col, func\_info in best\_functions.items():

45.         ideal\_func = func\_info['ideal\_function']

46.         y\_vals = ideal\_df[ideal\_func]

47.         plt.plot(x\_vals, y\_vals, color=colors[i % len(colors)],

48.                  linewidth=2, label=f"Chosen Ideal {ideal\_func} for {train\_col}")

49.         i += 1

50.     plt.scatter(test\_df['x'], test\_df['y'], color='black', label="Test Data")

51.     plt.xlabel("x")

52.     plt.ylabel("y")

53.     plt.title("Test Data and Chosen Ideal Functions")

54.     plt.legend()

55.     plt.grid(True)

56.     plt.show()

57.

#### 1.2.4 exceptions.py

1. class TableNotFoundError(Exception):

2.     """Raised when a specified table does not exist in the database."""

3.     def \_\_init\_\_(self, table\_name):

4.         super().\_\_init\_\_(f"Table '{table\_name}' does not exist in the database.")

5.

6. class ColumnMismatchError(Exception):

7.     """Raised when the columns in the CSV file do not match the table columns."""

8.     def \_\_init\_\_(self, table\_name):

9.         super().\_\_init\_\_(f"Column mismatch detected for table '{table\_name}'. Ensure the CSV headers match the table schema.")

10.

11. class DataInsertionError(Exception):

12.     """Raised when there is an error inserting data into the database."""

13.     def \_\_init\_\_(self, table\_name, message):

14.         super().\_\_init\_\_(f"Error inserting data into table '{table\_name}': {message}")

15.

#### 1.2.5 main.py

1. from modules import idealfunctionsmodule

2. from modules import plotdata

3. from modules import sqlite\_helper

4. import pandas as pd

5.

6. def get\_pd\_from\_csv(file\_path):

7.     """

8.     Load a CSV file into a pandas DataFrame.

9.

10.     Args:

11.         file\_path (str): Path to the CSV file.

12.

13.     Returns:

14.         pd.DataFrame: DataFrame containing the CSV data.

15.     """

16.     return pd.read\_csv(file\_path)

17.

18. def get\_data\_from\_source(source\_method=3, csv\_path\_prefix="data/", db\_path="data/data.db"):

19.     """

20.     Returns dataframes based on the specified data source method.

21.

22.     Args:

23.         source\_method (int): Method to get data:

24.             1 - Load directly from CSV files

25.             2 - Insert CSV data into SQLite and then retrieve it

26.             3 - Retrieve data from existing SQLite database (default)

27.         csv\_path\_prefix (str): Path prefix for CSV files (default: "data/")

28.         db\_path (str): Path to SQLite database (default: "data/data.db")

29.

30.     Returns:

31.         tuple: (ideal\_df, train\_df, test\_df) containing the datasets

32.     """

33.     if source\_method == 1:

34.         print("Loading data directly from CSV files...")

35.         ideal\_df = get\_pd\_from\_csv(f"{csv\_path\_prefix}ideal.csv")

36.         train\_df = get\_pd\_from\_csv(f"{csv\_path\_prefix}train.csv")

37.         test\_df = get\_pd\_from\_csv(f"{csv\_path\_prefix}test.csv")

38.     elif source\_method == 2:

39.         print("Inserting CSV data into SQLite and retrieving it...")

40.         sqlite\_helper.insert\_csv\_to\_table(f"{csv\_path\_prefix}ideal.csv", db\_path, "ideal")

41.         sqlite\_helper.insert\_csv\_to\_table(f"{csv\_path\_prefix}train.csv", db\_path, "train")

42.         sqlite\_helper.insert\_csv\_to\_table(f"{csv\_path\_prefix}test.csv", db\_path, "test")

43.         ideal\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "ideal")

44.         train\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "train")

45.         test\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "test")

46.     else:  # Default to option 3 or handle any invalid input

47.         print("Retrieving data from existing SQLite database...")

48.         ideal\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "ideal")

49.         train\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "train")

50.         test\_df = sqlite\_helper.get\_table\_data\_as\_df(db\_path, "test")

51.

52.     return ideal\_df, train\_df, test\_df

53.

54. def main():

55.     """

56.     Main function to execute the workflow:

57.     1. Load ideal, training, and test datasets.

58.     2. Fit ideal functions to training data.

59.     3. Choose the best ideal functions based on RMSE.

60.     4. Map test data to the best ideal functions.

61.     5. Generate and display plots for ideal, training, and test data.

62.     """

63.     # Get data using the default source method (3)

64.     ideal\_df, train\_df, test\_df = get\_data\_from\_source()

65.

66.     fitter = idealfunctionsmodule.FunctionFitter(ideal\_df, train\_df)

67.     fitter.fit\_ideal\_functions()

68.

69.     best\_functions = fitter.choose\_best\_ideal\_functions()

70.     print("Chosen ideal functions based on training RMSE:")

71.     for train\_col, info in best\_functions.items():

72.         print(f"{train\_col} -> Ideal Function: {info['ideal\_function']}, RMSE: {info['rmse']:.4f}")

73.

74.     mapper = idealfunctionsmodule.TestMapper(best\_functions, test\_df)

75.     mapping\_results = mapper.map\_test\_data()

76.     print("Test Data Mapping Results:")

77.     print(mapping\_results)

78.

79.     plotdata.plot\_ideal\_functions(ideal\_df)

80.     plotdata.plot\_training\_data(train\_df)

81.     plotdata.plot\_test\_data\_with\_ideal(test\_df, ideal\_df, best\_functions)

82.     print("Plots generated successfully.")

83.

84. if \_\_name\_\_ == "\_\_main\_\_":

85.     """

86.     Entry point of the script. Calls the main function.

87.     """

88.     main()

89.

#### 1.2.6 test\_all.py

1. import unittest

2. import pandas as pd

3. import math

4. from modules import idealfunctionsmodule

5.

6. class TestFunctionFitter(unittest.TestCase):

7.     """

8.     Unit tests for the FunctionFitter class, which fits ideal functions to training data

9.     and selects the best functions based on RMSE.

10.     """

11.

12.     def setUp(self):

13.         """

14.         Set up test data for FunctionFitter tests.

15.         Creates ideal and training DataFrames with known relationships.

16.         """

17.         self.ideal\_df = pd.DataFrame({

18.             'x': [0, 1, 2, 3, 4],

19.             'ideal1': [1, 3, 5, 7, 9],  # y = 2\*x + 1

20.             'ideal2': [0, 3, 6, 9, 12]  # y = 3\*x + 0

21.         })

22.

23.         self.train\_df = pd.DataFrame({

24.             'x': [0, 1, 2, 3, 4],

25.             'train1': [1, 3, 5, 7, 9],  # y = 2\*x + 1 (exact match with ideal1)

26.             'train2': [0, 4, 7, 10, 13]  # y = 3\*x + 1 (not an exact match with ideal2)

27.         })

28.

29.     def test\_fit\_and\_choose\_best\_functions(self):

30.         """

31.         Test the fit\_ideal\_functions and choose\_best\_ideal\_functions methods.

32.         Verifies that the ideal functions are fitted correctly and the best functions

33.         are chosen based on RMSE.

34.         """

35.         fitter = idealfunctionsmodule.FunctionFitter(self.ideal\_df, self.train\_df)

36.         fitter.fit\_ideal\_functions()

37.

38.         self.assertIn('ideal1', fitter.ideal\_models)

39.         self.assertIn('ideal2', fitter.ideal\_models)

40.

41.         # Checking if slope and intercept are calculated as expected

42.         ideal1\_model = fitter.ideal\_models['ideal1']

43.         ideal2\_model = fitter.ideal\_models['ideal2']

44.         self.assertAlmostEqual(ideal1\_model['slope'], 2, places=2)

45.         self.assertAlmostEqual(ideal1\_model['intercept'], 1, places=2)

46.         self.assertAlmostEqual(ideal2\_model['slope'], 3, places=2)

47.         self.assertAlmostEqual(ideal2\_model['intercept'], 0, places=2)

48.

49.         # Get the best functions for the training data.

50.         best\_functions = fitter.choose\_best\_ideal\_functions()

51.

52.         # For train1, the best ideal should be ideal1 (exact match).

53.         self.assertEqual(best\_functions['train1']['ideal\_function'], 'ideal1')

54.

55.         # For train2, the best ideal should be ideal2.

56.         self.assertEqual(best\_functions['train2']['ideal\_function'], 'ideal2')

57.

58. class TestTestMapper(unittest.TestCase):

59.     """

60.     Unit tests for the TestMapper class, which maps test data points to the best ideal functions

61.     based on minimum deviation.

62.     """

63.

64.     def setUp(self):

65.         """

66.         Set up test data for TestMapper tests.

67.         Creates ideal, training, and test DataFrames with known relationships.

68.         """

69.         self.ideal\_df = pd.DataFrame({

70.             'x': [0, 1, 2, 3, 4],

71.             'ideal1': [1, 3, 5, 7, 9],

72.             'ideal2': [0, 3, 6, 9, 12]

73.         })

74.         self.train\_df = pd.DataFrame({

75.             'x': [0, 1, 2, 3, 4],

76.             'train1': [1, 3, 5, 7, 9],

77.             'train2': [0, 4, 7, 10, 13]

78.         })

79.

80.         fitter = idealfunctionsmodule.FunctionFitter(self.ideal\_df, self.train\_df)

81.         fitter.fit\_ideal\_functions()

82.         self.best\_functions = fitter.choose\_best\_ideal\_functions()

83.

84.         # Create a test dataframe.

85.         # For x = 2:

86.         #   ideal1 predicts 2\*2+1 = 5, ideal2 predicts 3\*2+0 = 6.

87.         # For x = 3:

88.         #   ideal1 predicts 2\*3+1 = 7, ideal2 predicts 3\*3+0 = 9.

89.         # We choose y values to clearly favor one model over the other.

90.         self.test\_df = pd.DataFrame({

91.             'x': [2, 3],

92.             'y': [5, 9]  # Row 1 should map to ideal1, row 2 should map to ideal2.

93.         })

94.

95.     def test\_map\_test\_data(self):

96.         """

97.         Test the map\_test\_data method to ensure correct mapping of test data to ideal functions.

98.         Verifies that the assigned functions and deviations are calculated correctly.

99.         """

100.         mapper = idealfunctionsmodule.TestMapper(self.best\_functions, self.test\_df)

101.         mapping\_results = mapper.map\_test\_data()

102.

103.         # Check that the results contain the expected columns.

104.         self.assertTrue('x' in mapping\_results.columns)

105.         self.assertTrue('y' in mapping\_results.columns)

106.         self.assertTrue('assigned\_function' in mapping\_results.columns)

107.         self.assertTrue('deviation' in mapping\_results.columns)

108.

109.         # Verify that there are exactly two rows.

110.         self.assertEqual(len(mapping\_results), 2)

111.

112.         # For the first row (x = 2, y = 5), ideal1 should be chosen.

113.         row1 = mapping\_results.iloc[0]

114.         self.assertEqual(row1['assigned\_function'], 'ideal1')

115.         self.assertAlmostEqual(row1['deviation'], 0, places=5)

116.

117.         # For the second row (x = 3, y = 9), ideal2 should be chosen.

118.         row2 = mapping\_results.iloc[1]

119.         self.assertEqual(row2['assigned\_function'], 'ideal2')

120.         self.assertAlmostEqual(row2['deviation'], 0, places=5)

121.

122. if \_\_name\_\_ == '\_\_main\_\_':

123.     """

124.     Run all unit tests.

125.     """

126.     unittest.main()

127.

1. Ion Mobility Spectrometry: Fundamental Concepts, Instrumentation, Applications, and the Road Ahead (<https://pmc.ncbi.nlm.nih.gov/articles/PMC6832852/>) [↑](#footnote-ref-1)
2. https://www.analyticon.eu/en/ion-mobility-spectrometry.html [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/Ion\_mobility\_spectrometry#Ion\_mobility [↑](#footnote-ref-3)
4. Channels are the sensor values that are measured by the multiple sensors present during IMS measurement. In this project, we will be using an 8 channel IMS measurement. [↑](#footnote-ref-4)
5. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol\_filter.html [↑](#footnote-ref-5)
6. https://www.statsmodels.org/v0.10.2/examples/notebooks/generated/predict.html [↑](#footnote-ref-6)