mcetl Documentation

Release 0.4.1

Donald Erb

Jan 26, 2021

CONTENTS:

mcetl

materials characterization

extract transform load

mcetl is a small-scale, GUI-based Extract-Transform-Load framework focused on materials characterization.

- For Python 3.7+
- Open Source: BSD 3-Clause License
- Source Code: https://github.com/derb12/mcetl
- Documentation: https://mcetl.readthedocs.io.

mcetl is focused on reducing the time required to repeatedly process data files and write the results to Excel. It does this by allowing the user to define DataSource objects for each separate source of data. Each DataSource contains information such as the options needed to import data from files, the calculations that will be performed on the data, and the options for writing the data to Excel. Once a DataSource is created, it can be selected within mcetl's main user interface.

In addition, mcetl provides fitting and plotting user interfaces that can be used without any prior setup.

CHAPTER

ONE

INTRODUCTION

1.1 Purpose

The aim of mcetl is to ease the repeated processing of data files. Contrary to its name, mcetl can process any tabulated files (txt, csv, tsv, xlsx, etc.), and does not require that the files originate from materials characterization. However, the focus on materials characterization was selected because:

- Most data files from materials characterization are relatively small in size (a few kB or MB).
- Materials characterization files are typically cleanly tabulated and do not require handling messy or missing data.
- It was the author's area of usage and naming things is hard...

mcetl requires only a very basic understanding of Python to use, and allows a single person to create a tool that their entire group can use to process data and produce Excel files with a consistent style. mcetl can create new Excel files when processing data or saving peak fitting results, or it can append to an existing Excel file to easily work with already created files.

1.2 Limitations

- Data from files is fully loaded into memory for processing, so mcetl is not suited for processing files whose total memory size is large (e.g. cannot load a 10 GB file on a computer with only 8 GB of RAM).
- mcetl does not provide any resources for processing data files directly from characterization equipment (such as .XRDML, .PAR, etc.). Other libraries such as xylib[1] already exist and are capable of converting many such files to a format mcetl can use (txt, csv, etc.).
- The peak fitting and plotting modules in mcetl are not as feature-complete as other alternatives such as Origin[2], fityk[3], SciDAVis[4], etc. The modules are included in mcetl in case those better alternatives are not available, and the author highly recommends using those alternatives over mcetl if available.

[1] https://github.com/wojdyr/xylib
[2] https://originlab.com
[3] https://fityk.nieto.pl
[4] https://sourceforge.net/projects/scidavis/

CHAPTER

TWO

INSTALLATION

2.1 Dependencies

mcetl requires Python[5] version 3.7 or later and the following libraries:

- asteval[6]
- lmfit[7] (>= 1.0)
- Matplotlib[8] (>= 3.1)
- NumPy[9] (>= 1.8)
- openpyxl[10] (>= 2.4)
- pandas[11] (>= 0.25)
- PySimpleGUI[12] (>= 4.29)
- SciPy[13]

All of the required libraries should be automatically installed when installing mcetl using either of the two installation methods below.

Additionally, mcetl can optionally use Pillow[14] to allow for additional options when saving figures in the plotting GUI.

2.2 Stable Release

mcetl is easily installed from pypi[15] using pip[16], simply by running the following command in your terminal:

```
pip install --upgrade mcetl
```

[5] https://python.org
[6] https://github.com/newville/asteval
[7] https://lmfit.github.io/lmfit-py/
[8] https://matplotlib.org
[9] https://numpy.org
[10] https://openpyxl.readthedocs.io/en/stable/
[11] https://pandas.pydata.org
[12] https://github.com/PySimpleGUI/PySimpleGUI
[13] https://www.scipy.org/scipylib/index.html
[14] https://python-pillow.org/
[15] https://pypi.org/project/mcetl
[16] https://pip.pypa.io

This is the preferred method to install mcetl, as it will always install the most recent stable release. Note that the `--upgrade` tag is used to ensure that the most recent version of mcetl is downloaded and installed, even if an older version is currently installed.

2.3 Development Version

The sources for mcetl can be downloaded from the Github repo[17].

The public repository can be cloned using:

```
git clone https://github.com/derb12/mcetl.git
```

Once the repository is downloaded, it can be installed with:

```
cd mcetl
python setup.py install
```

[17] https://github.com/derb12/mcetl

CHAPTER

THREE

QUICK START

The sections below give a quick introduction to using mcetl, requiring no setup. For a more detailed introduction, refer to the *tutorials section* (page 9) of the documentation.

Note: On Windows operating systems, the GUIs can appear blurry due to how dpi scaling is handled. To fix, simply do:

```
import mcetl
mcetl.set_dpi_awareness()
```

The above code **must** be called before opening any GUIs, or else the dpi scaling will be incorrect.

3.1 Main GUI

The main GUI for mcetl contains options for processing data, fitting, plotting, writing data to Excel, and moving files.

Before using the main GUI, DataSource objects must be created. Each DataSource contains the information for reading files for that DataSource (such as what separator to use, which rows and columns to use, labels for the columns, etc.), the calculations that will be performed on the data, and the options for writing the data to Excel (formatting, placement in the worksheet, etc.).

The following will create a DataSource named 'tutorial' with the default settings, and will then open the main GUI.

```
import mcetl

simple_datasource = mcetl.DataSource(name='tutorial')
mcetl.launch_main_gui([simple_datasource])
```

3.2 Fitting Data

To use the fitting module in mcetl, simply do:

```
from mcetl import fitting
fitting.launch_fitting_gui()
```

A window will then appear to select the data file(s) to be fit and the Excel file for saving the results. No other setup is required for doing fitting.

After doing the fitting, the fit results and plots will be saved to Excel.

3.3 Plotting

To use the plotting module in mcetl, simply do:

```
from mcetl import plotting
plotting.launch_plotting_gui()
```

Similar to fitting, a window will then appear to select the data file(s) to be plotted, and no other setup is required for doing plotting.

When plotting, the image of the plots can be saved to all formats supported by Matplotlib[18], including tiff, jpg, png, svg, and pdf.

In addition, the layout of the plots can be saved to apply to other figures later, and the data for the plots can be saved so that the entire plot can be recreated.

To reopen a figure saved through mcetl, do:

```
plotting.load_previous_figure()
```

[18] https://matplotlib.org

CHAPTER

FOUR

TUTORIALS

Tutorials are available for using the main, fitting, and plotting GUIs, as well as for importing data, changing the style of the Excel output, and generating example raw data.

4.1 Main GUI

The Main GUI for mcetl contains options for processing data, fitting data, plotting data, writing data to Excel, and moving files. Before using the Main GUI, DataSource and Function objects must be created.

Each *DataSource* (page 54) object contains the information for reading files for that DataSource (such as what separator to use, which rows and columns to use, labels for the columns, etc.), the calculations that will be performed on the data, and the options for writing the data to Excel (formatting, placement in the worksheet, etc.).

Function objects come in three types:

- *PreprocessFunction* (page 65): process each data entry as soon as it is imported from the files
- *CalculationFunction* (page 64): process each entry in each sample in each dataset
- *SummaryFunction* (page 65): process each sample or each dataset; performed last

Naming Conventions

Within the tutorials, data is sectioned into datasets, samples, and entries. This is done to help understand the grouping of data. A dataset can be considered a single, large grouping of data whose contents are all related somehow. If writing to Excel, each dataset will occupy its own sheet. A sample is a grouping within a dataset, and can contain multiple data entries. Each entry can be considered as the contents of a single file, but could be broken down further if desired.

For example, consider a study of the effects of processing temperature and carbon percentage on the mechanical properties of steel. If there are two processing temperatures (700°C and 800°C) five different carbon amounts (0 wt%, 1 wt%, 2 wt%, 3 wt%, and 4 wt%), and three measurements per sample, then the following grouping could be made:

- Dataset 1 (700°C)
  - Sample 1 (0 wt% carbon)
    - Entry 1 (first measurement file)
    - Entry 2 (second measurement file)
    - Entry 3 (third measurement file)
  - Sample 2 (1 wt% carbon)
    - Entry 1 (first measurement file)
    - Entry 2 (second measurement file)
    - Entry 3 (third measurement file)

    - Sample 3 etc...
- Dataset 2 (800°C)
    - Sample 1 (0 wt% carbon)
        * Entry 1 (first measurement file)
        * Entry 2 (second measurement file)
        * Entry 3 (third measurement file)
    - Sample 2 etc...

4.1.1 Creating Function Objects

Function objects are simple wrappers around functions that allow referencing the function by a name, specifying which columns the function should act on, and other relevant information about the function's behavior. Function objects come in three types:

- PreprocessFunctions: process each data entry as soon as it is imported from the files
- CalculationFunctions: process each entry in each sample in each dataset
- SummaryFunctions: process each sample or each dataset; performed last

Note that all functions will process pandas DataFrames[19]. To speed up calculations within the functions, it is suggested to make use of vectorization of the pandas DataFrame or Series or of the underlying numpy arrays, since vectorization can be significantly faster than iterating over each index using a for loop.

This tutorial will cover some basic examples for the three Function types. To see more advanced examples, see the example programs[20] in the GitHub respository. The use_main_gui.py example program shows many different uses of the three Function objects, and the creating_functions.py example program walks through the internals of how *launch_main_gui()* (page 67) calls the Functions so that each individual step can be understood.

Note: For functions of all three Function types, it is a good idea to add `**kwargs` to the function input. In later versions of mcetl, additional items may be passed to functions, but they will always be added as keyword arguments (ie. passed as `name=value`). By adding `**kwargs`, any unwanted keyword arguments for functions will be ignored and not cause issues when upgrading mcetl.

PreprocessFunctions

A *PreprocessFunction* (page 65) will perform its function on each data entry individually. Example usage of a PreprocessFunction can be to split a single data entry into multiple entries, collect information on all data entries in each sample in each dataset for usage later, or just simple processes that are easier to do per entry rather than when each dataset is grouped together.

The function for a PreprocessFunction must take at least two arguments: the dataframe containing the data for the entry, and a list of indices that tell which columns of the dataframe contain the data used by the function. **The function must return a list of dataframes after processing**, even if only one dataframe is used within the function.

A simple function to split dataframes based on the segment number, and then remove the segment number column is shown below.

[19] https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html
[20] https://github.com/derb12/mcetl/tree/main/examples

```
import mcetl
import numpy as np

def split_segments(df, target_indices, **kwargs):
    """
    Preprocess function that separates each entry based on the segment number.

    Also removes the segment column after processing since it is not needed
    in the final output.

    Parameters
    ----------
    df : pandas.DataFrame
        The dataframe of the entry.
    target_indices : list(int)
        The indices of the target columns.

    Returns
    -------
    output_dataframes : list(pandas.DataFrame)
        The list of dataframes after splitting by segment number.

    """

    segment_index = target_indices[0]
    segment_col = df[df.columns[segment_index]].to_numpy()
    mask = np.where(segment_col[:-1] != segment_col[1:])[0] + 1 # + 1 since mask␣
↪loses one index

    output_dataframes = np.array_split(df, mask)

    for dataframe in output_dataframes:
        dataframe.drop(segment_index, 1, inplace=True)

    return output_dataframes

# targets the 'segment' column from the imported data
segment_separator = mcetl.PreprocessFunction(
    name='segment_separator', target_columns='segment',
    function=split_segments, deleted_columns=['segment']
)
```

In addition, PreprocessFunctions can be used for procedures that are easier to perform on each data entry separately, rather than when all of the data is collected together. For example, sorting each entry based on the values in one of its columns.

```
def sort_columns(df, target_indices, **kwargs):
    """Sorts the dataframe based on the values of the target column."""
    return [df.sort_values(target_indices[0])]

# targets the 'diameter' column from the imported data
pore_preprocess = mcetl.PreprocessFunction('sort_data', 'diameter', sort_columns)
```

CalculationFunctions

A *CalculationFunction* (page 64) will perform its function on each merged dataset. Each merged dataset is composed of all data entries for each sample concatenated together, resembling how the dataset will look when written to an Excel sheet. This makes the functions more difficult to create since target columns are given as nested lists of lists for each sample, but allows access to all data in a dataset, if required for complex calculations.

Each CalculationFunction can have two functions: one for performing the calculations on data for Excel and one for performing calculations on data for Python. This way, the Excel calculations can create strings to match Excel-specific functions, like using '=SUM(A1:A3)' to make the data dynamic within the Excel workbook, while the Python functions can calculate the actual numerical data (eg. sum(data[0:2]) to match the previous Excel formula). If only a single, numerical calculation is desired, regardless of whether the data is being output to Excel or Python, then only a single function needs to be specified (an example of such a function is given below).

The functions for a CalculationFunction must take at least three arguments: the dataframe containing the data for the dataset, a list of lists of integers that tell which columns in the dataframe contain the data used by the function, and a list of lists of integers that tell which columns in the dataframe are used for the output of the function. Additionally, two keyword arguments are passed to the function: `excel_columns`, which is a list of strings corresponding to the columns in Excel used by the dataset (eg. ['A', 'B', 'C', 'D']) if doing Excel functions and is None if doing Python functions, and `first_row`, which is an integer telling the first row in Excel that the data will begin on (3 by default, since the first row is the sample name and the second is the column name). **The functions must return a DataFrame after processing**, even if all changes to the input DataFrame were done in place.

A simple function that adds a column of offset data is shown below.

```
import mcetl
import numpy as np

def offset_data_excel(df, target_indices, calc_indices, excel_columns,
                      first_row, offset, **kwargs):
    """Creates a string that will add an offset to data in Excel."""

    total_count = 0
    for i, sample in enumerate(calc_indices):
        for j, calc_col in enumerate(sample):
            y = df[target_indices[0][i][j]]
            y_col = excel_columns[target_indices[0][i][j]]
            calc = [
                f'={y_col}{k + first_row} + {offset * total_count}' for k in␣
↪range(len(y))
            ]
            # use np.where(~np.isnan(y)) so that the calculation works for unequally-
↪sized
            # datasets
            df[calc_col] = np.where(~np.isnan(y), calc, None)
            total_count += 1

    return df

def offset_data_python(df, target_indices, calc_indices, first_row, **kwargs):
    """Adds an offset to data."""

    total_count = 0
    for i, sample in enumerate(calc_indices):
        for j, calc_col in enumerate(sample):
            y = df[target_indices[0][i][j]]
            df[calc_col] = y + (kwargs['offset'] * total_count)
            total_count += 1
```

(continues on next page)

(continued from previous page)

```
    return df

# targets the 'data' column from the imported data
offset = mcetl.CalculationFunction(
    name='offset', target_columns='data', functions=(offset_data_excel, offset_data_
↪python),
    added_columns=1, function_kwargs={'offset': 10}
)
```

Alternatively, the two functions could be combined into one, and the calculation route could be decided by examining the value of the `excel_columns` input, which is a list of strings if processing for Excel, and None when processing for Python.

```
import mcetl
import numpy as np

def offset_data(df, target_indices, calc_indices, excel_columns,
                first_row, offset, **kwargs):
    """Adds an offset to data."""

    total_count = 0
    for i, sample in enumerate(calc_indices):
        for j, calc_col in enumerate(sample):
            if excel_columns is not None: # do Excel calculations
                y = df[target_indices[0][i][j]]
                y_col = excel_columns[target_indices[0][i][j]]
                calc = [
                    f'= {y_col}{k + first_row} + {offset * total_count}' for k in␣
↪range(len(y))
                ]
                df[calc_col] = np.where(~np.isnan(y), calc, None)

            else: # do Python calculations
                y = df[target_indices[0][i][j]]
                df[calc_col] = y + (offset * total_count)
            total_count += 1

    return df

# targets the 'data' column from the imported data
offset = mcetl.CalculationFunction(
    name='offset', target_columns='data', functions=offset_data,
    added_columns=1, function_kwargs={'offset': 10}
)
```

To modify the contents of an existing column, the input for `added_columns` for CalculationFunction should be a string designating the target: either a variable from the imported data, or the name of a CalculationFunction.

```
import mcetl
import numpy as np

def normalize(df, target_indices, calc_indices, excel_columns, first_row, **kwargs):
    """Performs a min-max normalization to bound values between 0 and 1."""

    for i, sample in enumerate(calc_indices):
```

(continues on next page)

(continued from previous page)

```
        for j, calc_col in enumerate(sample):
            if excel_columns is not None:
                y = df[target_indices[0][i][j]]
                y_col = excel_columns[target_indices[0][i][j]]
                end = y.count() + 2
                calc = [
                    (f'=({y_col}{k + first_row} - MIN({y_col}$3:{y_col}${end})) / '
                     f'(MAX({y_col}$3:{y_col}${end}) - MIN({y_col}$3:{y_col}${end}))')
                    for k in range(len(y))
                ]

                df[calc_col] = np.where(~np.isnan(y), calc, None)

            else:
                y_col = df.columns[target_indices[0][i][j]]
                min_y = df[y_col].min()
                max_y = df[y_col].max()

                df[calc_col] = (df[y_col] - min_y) / (max_y - min_y)

    return df

def offset_normalized(df, target_indices, calc_indices, excel_columns,
                      offset, **kwargs):
    """Adds an offset to normalized data."""

    total_count = 0
    for i, sample in enumerate(calc_indices):
        for j, calc_col in enumerate(sample):
            y_col = df[target_indices[0][i][j]]
            offset_amount = offset * total_count
            if excel_columns is not None:
                df[calc_col] = y_col + f' + {offset_amount}'
            else:
                df[calc_col] = y_col + offset_amount
            total_count += 1

    return df

# targets the 'data' column from the imported data
normalize_func = mcetl.CalculationFunction(
    name='normalize', target_columns='data',
    functions=normalize, added_columns=1
)
# targets the 'normalize' column from the the 'normalize' CalculationFunction
# and also alters its contents
offset_func = mcetl.CalculationFunction(
    name='offset', target_columns='normalize', functions=offset_normalized,
    added_columns='normalize', function_kwargs={'offset': 10}
)
```

If the CalculationFunction does the same calculation, regardless of whether the data is going to Excel or for later processing in Python, then a mutable object, like a list, can be used in function_kwargs to signify that the calculation has been performed to prevent processing twice.

```
import mcetl
```

(continues on next page)

(continued from previous page)

```
def offset_numerical(df, target_indices, calc_indices, excel_columns, **kwargs):
    """Adds a numerical offset to data."""

    # Add this section to prevent doing numerical calculations twice.
    if excel_columns is None and kwargs['processed'][0]:
        return df # return to prevent processing twice
    elif excel_columns is not None:
        kwargs['processed'][0] = True

    # Regular calculation section
    offset = kwargs['offset']
    total_count = 0
    for i, sample in enumerate(calc_indices):
        for j, calc_col in enumerate(sample):
            df[calc_col] = df[target_indices[0][i][j]] + (offset * total_count)
            total_count += 1

    return df

# targets the 'data' column from the imported data
numerical_offset = mcetl.CalculationFunction(
    name='numerical offset', target_columns='data', functions=offset_numerical,
    added_columns=1, function_kwargs={'offset': 10, 'processed': [False]}
)
```

SummaryFunctions

A *SummaryFunction* (page 65) is very similar to CalculationFunctions, performing its functions on each merged dataset and requiring outputting a single DataFrame. However, SummaryFunctions differ from CalculationFunctions in that their added columns are not within the data entries themselves. Instead, SummaryFunctions can either be a sample SummaryFunction (by using `sample_summary=True` when creating the object), which is equivalent to appending a data entry to each sample in each dataset, or a dataset SummaryFunction (by using `sample_summary=False` when creating the object), which is equivalent to appending a sample to each dataset.

For example, consider calculating the elatic modulus from tensile tests. Each sample in the dataset will have multiple measurements/entries, so a sample SummaryFunction could be used to calculate the average elastic modulus for each sample, and a dataset SummaryFunction could be used to create a table listing the average elastic modulus for each sample in the dataset for easy referencing.

```
import mcetl
import numpy as np
import pandas as pd
from scipy import optimize

def stress_model(strain, modulus):
    """
    The linear estimate of the stress-strain curve using the strain and estimated␣
↪modulus.

    Parameters
    ----------
    strain : array-like
        The array of experimental strain values, unitless (with cancelled
        units, such as mm/mm).
```

(continues on next page)

(continued from previous page)

```
    modulus : float
        The estimated elastic modulus for the data, with units of GPa (Pa * 10**9).

    Returns
    -------
    array-like
        The estimated stress data following the linear model, with units of Pa.

    """
    return strain * modulus * 1e9


def tensile_calculation(df, target_indices, calc_indices, excel_columns, **kwargs):
    """Calculates the elastic modulus from the stress-strain curve for each entry."""

    if excel_columns is None and kwargs['processed'][0]:
        return df # return to prevent processing twice
    elif excel_columns is not None:
        kwargs['processed'][0] = True

    num_columns = 2 # the number of calculation columns per entry
    for i, sample in enumerate(calc_indices):
        for j in range(len(sample) // num_columns):
            strain_index = target_indices[0][i][j]
            stress_index = target_indices[1][i][j]
            nan_mask = (~np.isnan(df[strain_index])) & (~np.isnan(df[stress_index]))

            # to convert strain from % to unitless
            strain = df[strain_index].to_numpy()[nan_mask] / 100
             # to convert stress from MPa to Pa
            stress = df[stress_index].to_numpy()[nan_mask] * 1e6

            # only use data where stress varies linearly with respect to strain
            linear_mask = (
                (strain >= kwargs['lower_limit']) & (strain <= kwargs['upper_limit'])
            )
            initial_guess = 80 # initial guess of the elastic modulus, in GPa

            modulus, covariance = optimize.curve_fit(
                stress_model, strain[linear_mask], stress[linear_mask],
                p0=[initial_guess]
            )

            df[sample[0 + (j * num_columns)]] = pd.Series(('Value', 'Standard Error'))
            df[sample[1 + (j * num_columns)]] = pd.Series(
                (modulus[0], np.sqrt(np.diag(covariance)[0]))
            )

    return df


def tensile_sample_summary(df, target_indices, calc_indices, excel_columns, **kwargs):
    """Summarizes the mechanical properties for each sample."""

    if excel_columns is None and kwargs['processed'][0]:
        return df # to prevent processing twice
```

(continues on next page)

(continued from previous page)

```
    num_cols = 2 # the number of calculation columns per entry from tensile_
↪calculation
    for i, sample in enumerate(calc_indices):
        if not sample: # skip empty lists
            continue

        entries = []
        for j in range(len(target_indices[0][i]) // num_cols):
            entries.append(target_indices[0][i][j * num_cols:(j + 1) * num_cols])

        df[sample[0]] = pd.Series(['Elastic Modulus (GPa)'])
        df[sample[1]] = pd.Series([np.mean([df[entry[1]][0] for entry in entries])])
        df[sample[2]] = pd.Series([np.std([df[entry[1]][0] for entry in entries])])

    return df


def tensile_dataset_summary(df, target_indices, calc_indices, excel_columns,␣
↪**kwargs):
    """Summarizes the mechanical properties for each dataset."""

    if excel_columns is None and kwargs['processed'][0]:
        return df # to prevent processing twice

    # the number of samples is the number of lists in calc_indices - 1
    num_samples = len(calc_indices[:-1])

    # calc index is -1 since only the last dataframe is the dataset summary dataframe
    df[calc_indices[-1][0]] = pd.Series(
        [''] + [f'Sample {num + 1}' for num in range(num_samples)]
    )
    df[calc_indices[-1][1]] = pd.Series(
        ['Average'] + [df[indices[1]][0] for indices in target_indices[0][:-1]]
    )
    df[calc_indices[-1][2]] = pd.Series(
        ['Standard Deviation'] + [df[indices[2]][0] for indices in target_
↪indices[0][:-1]]
    )

    return df

# share the keyword arguments between all function objects
tensile_kwargs = {'lower_limit': 0.0015, 'upper_limit': 0.005, 'processed': [False]}

# targets the 'data' column from the imported data
tensile_calc = mcetl.CalculationFunction(
    name='tensile calc', target_columns=['strain', 'stress'],
    functions=tensile_calculation, added_columns=2,
    function_kwargs=tensile_kwargs
)
# targets the columns from the the 'tensile calc' CalculationFunction
stress_sample_summary = mcetl.SummaryFunction(
    name='tensile sample summary', target_columns=['tensile calc'],
    functions=tensile_sample_summary, added_columns=3,
    function_kwargs=tensile_kwargs, sample_summary=True
)
# targets the columns from the the 'tensile sample summary' SummaryFunction
```

(continues on next page)

(continued from previous page)

```
stress_dataset_summary = mcetl.SummaryFunction(
    name='tensile dataset summary', target_columns=['tensile sample summary'],
    functions=tensile_dataset_summary, added_columns=3,
    function_kwargs=tensile_kwargs, sample_summary=False
)
```

4.1.2 Creating DataSources

DataSource (page 54) objects are the main controlling object when using mcetl's main GUI.

DataSource Inputs

This section groups the inputs for DataSource by their usage, and explains possible inputs. The only necessary input for a DataSource is its `name`, which is displayed in the GUI when selecting which DataSource to use.

Note that most inputs for DataSource will supply defaults for entering information into the GUIs, and can be overwritten when using the main GUI.

Sections

Using Raw Data Files

File Searching

The keyword `file_type` should be a string of the file extension of data files for the DataSource, such as 'csv' or 'txt'. When searching for files, either manually or using a keyword search, the file extension is used to limit the number of files to select.

The keyword `num_files` should be an interger telling how many files should be associated for each sample for the DataSource. Only used if using a keyword search for files.

Importing Data

`column_numbers` should be a list of integers telling which columns to import from the raw data file. The order of columns follows the ordering within column_numbers, so [0, 1, 2] would import the first, second, and third columns from the data file and keep their order, while [1, 2, 0] would still import the three columns but rearrange them so that the first column is now the third column.

`start_row` and `end_row` should be integers telling which row in the raw data file should be the first and last row, respectively. Note that end_row counts up from the bottom of the data file, so that end_row=0 would be the last row, end_row=1 would be the second to last row, etc.

`separator` should be a string designating the separator used in the raw data file to separate columns. For example, comma-separated files (csv) have ',' as the separator.

See the *Importing Data section* (page 26) of the tutorial for more in depth discussion on importing data.

Processing Data

Functions

The keyword `functions` should be a list or tuple of all the PreprocessFunctions, CalculationFunctions, and/or SummaryFunctions that are needed for processing data for the DataSource. PreprocessFunctions will be performed first, followed by CalculationFunctions and then SummaryFunctions. For each Function type, the order in which the functions are performed is according to their ordering in the functions input.

The keyword `unique_variables` should be a list of strings designating the names to give to columns that contain necessary data. The Function objects of the DataSource will then be able set their target_columns to those columns. For example, in the previous tutorial section on creating Function objects, the example Function objects had target_columns such as 'data' and 'diameter' from imported data. The 'data' and 'diameter' columns would be the DataSource's unique_variables.

`unique_variable_indices` should be a list of intergers telling the indices of the columns within column_numbers belong to the unique_variables. For example, if column_numbers was [1, 2, 0], and Column 2 contained the data for the unique variable, then unique_variable_indices would be [1] since Column 2 is at index 1 within column_numbers.

Column Names

The keyword `column_labels` should be a list of strings, designating the default column labels for a single data entry. The column labels should accomodate both the names of the imported data columns, and all of the columns added by CalculationFunctions and SummaryFunctions.

To aid setting up the column_labels, a DataSource instance has the method *print_column_labels_template()* (page 57), which will tell how many labels belong to imported data, CalculationFunctions, sample-summary SummaryFunctions, and dataset-summary SummaryFunctions, and give a template for the column_labels input.

The example below shows creating a DataSource for tensile testing, and uses the tensile_calc, stress_sample_summary, stress_dataset_summary Function objects created in the previous tutorial section on creating Function objects.

```
import mcetl

tensile = mcetl.DataSource(
    'Tensile Test',
    functions=[tensile_calc, stress_sample_summary, stress_dataset_summary],
```

(continues on next page)

(continued from previous page)

```
    unique_variables=['stress', 'strain'],
    column_numbers=[4, 3, 0, 1, 2],
    unique_variable_indices=[1, 0],
    # column_labels only currently has labels for the imported data, not the Functions
    column_labels=['Strain (%)', 'Stress (MPa)', 'Time (s)', 'Extension (mm)', 'Load
↪(kN)'],
)

tensile.print_column_labels_template()
```

The `label_entries` keyword can be set to True to append a number to the column labels of each entry in a sample if there is more than one entry. For example, the column label 'data' would become 'data, 1', 'data, 2', etc.

Figure Appearance

The `figure_rcparams` keyword can be used to set the style of figures for fitting and plotting using Matplotlib's rcParams[21]. The input should be a dictionary, like below:

```
example_rcparams = {
    'font.serif': 'Times New Roman',
    'font.family': 'serif',
    'font.size': 12,
    'xtick.direction': 'in',
    'ytick.direction': 'in'
}
```

Excel Output

Location within the Workbook

Where data is placed within the output Excel file is determined by the `excel_column_offset` and `excel_row_offset` keywords, which expect integers. By default, data is placed in the workbook starting at cell A1. To change the starting cell to D8, for example, the excel_column_offset would be 3 and the excel_row_offset would be 7.

Spacing between Samples and Entries

To place empty columns in the Excel workbook to help visually separate the various samples and entries, use `entry_separation` and `sample_separation`. For example, to place one empty columns between each entry in a sample, and two empty columns between each sample in a dataset, use entry_separation=1 and sample_separation=2, respectively.

[21] https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams

Excel Plot Options

The `xy_plot_indices` keyword should be a list with two integers, telling the indices of the columns to use for the x- and y-axes when plotting each entry in Excel. Note that the the indices can refer to the columns from imported data or from columns added by CalculationFunctions. For example, if three columns were imported, and two additional columns were added by CalculationFunctions, then the total column indices would be [0, 1, 2, 3, 4] and plot indices could be [0, 1] or [1, 4], etc.

Sample SummaryFunctions and dataset SummaryFunctions are also allowed to be plotted in Excel and will use the xy_plot_indices.

Specifying Excel Styles

The styles used in the output Excel file are described by using the `excel_writer_styles` keyword. See the *Excel Styles section* (page 30) of the tutorial for discussion on how to change the style of output Excel file.

Examples

The examples below show creating DataSources using some of the keyword inputs discussed above as well as the Function objects created in the previous tutorial section on creating Function objects.

```
import mcetl

xrd = mcetl.DataSource(
    'X-ray Diffraction',
    column_labels=['2\u03B8 (\u00B0)', 'Intensity (Counts)', 'Offset Intensity (a.u.)
↪'],
    functions=[offset],
    column_numbers=[1, 2],
    unique_variables=['data'],
    unique_variable_indices=[1],
    start_row=1,
    end_row=0,
    separator=',',
    xy_plot_indices=[0, 2],
    file_type='csv',
    num_files=1,
    entry_separation=1,
    sample_separation=2,
)

tensile = mcetl.DataSource(
    'Tensile Test',
    functions=[tensile_calc, stress_sample_summary, stress_dataset_summary],
    column_numbers=[4, 3, 0, 1, 2],
    unique_variables=['stress', 'strain'],
    unique_variable_indices=[1, 0],
    xy_plot_indices=[0, 1],
    column_labels=[
        'Strain (%)', 'Stress (MPa)', 'Time (s)', 'Extension (mm)', 'Load (kN)', #␣
↪imported data
        '', 'Elastic Modulus (GPa)', # CalculationFunction
        'Property', 'Average', 'Standard Deviation', # sample SummaryFunction
        'Sample', 'Elastic Modulus (GPa)', '' # dataset SummaryFunction
    ],
```

(continues on next page)

(continued from previous page)

```
    start_row=6,
    separator=',',
    file_type='txt',
    num_files=3,
    entry_separation=2,
    sample_separation=3
)
```

4.1.3 Using the Main GUI

Once all of the desired DataSource objects are created, simply group the DataSource objects together in a list or tuple, and then call *launch_main_gui()* (page 67). For example, using the two DataSource objects created from the last section gives:

```
import mcetl

all_datasources = (xrd, tensile)
mcetl.launch_main_gui(all_datasources)
```

The main GUI has the following steps:

- Main Menu (Choosing the DataSource and Processing Steps)
- File Selection
- Importing Data from Files
- Naming Samples & Columns
- Processing

Main Menu

In the main menu, the desired DataSource, the file selection method, and any processing steps are selected.

The possible processing steps are:

- Process the data using the Function objects for the selected DataSource
- Fit the data
- Plot the data
- Save the data to Excel
- Move files

Note that if the only selected processing step is moving files, then no data is actually imported.

File Selection

Manually Selecting Files

Manual selection of files is easy and fast. To start, the number of datasets needs to be entered, and then the number of samples per dataset. Each dataset will be one sheet when writing to Excel.

Next, a window will appear to select the files for each sample in each dataset (see the file selection window in the *gallery section* (page 85) of the documentation). Simply press 'Add Files', and select all of the files for each sample. To remove a file, just select it and press 'Del Files'.

Searching Using Keywords

Files can also be searched for using keywords. However, this is usually more difficult than manually selecting the files and should only be used if searching for files within a deeply nested folder structure. Further explanation of keyword searching for files is given when actually using within the main GUI, and will not be covered here since its usage is discouraged.

Note: After selecting files, a file called "previous_files_{DataSource.name}.json" is locally saved, where {DataSource.name} is the name of the selected DataSource. This allows quickly bypassing file selection in case of repeated processing of the same files.

To get the file path where the files are saved, simply do:

```
print(str(mcetl.main_gui.SAVE_FOLDER))
```

Importing Data from Files

See the *Importing Data section* (page 26) of the tutorial for the explanation of how mcetl imports raw data.

Note that numeric data after importing is downcast to the lowest possible representation (for example float is converted to numpy.float32) to reduce memory usage. If this would be an issue, be sure to include appropriate Function objects to convert to the desired dtype.

Naming Samples & Columns

The name of each sample for each dataset can be specified, as well as the column name for each column in the dataset. Column names are generated using the `column_names` input for the selected DataSource. The *gallery section* (page 85) of the documentation shows an example of the sample and column naming window.

Processing

Processing includes any processing steps selected in the main menu, including processing the data using the Function objects for the selected DataSource, fitting the data, plotting the data, writing to Excel, or moving files. Each step should be self-explanatory.

The output of the launch_main_gui function will be a single dictionary with the following keys and values:

'dataframes': list or None A list of lists of pandas.DataFrame, with each dataframe containing the data imported from a raw data file; will be None if the function fails before importing data, or if the only processing step taken was moving files.

'fit_results': list or None A nested list of lists of lmfit ModelResult objects, with each ModelResult pertaining to the fitting of a data entry, each list of ModelResults containing all of the fits for a single sample, and east list of lists pertaining to the data within one dataset. Will be None if fitting is not done, or only partially filled if the fitting process ends early.

'plot_results': list or None A list of lists, with one entry per dataset. Each interior list is composed of a matplotlib.Figure object and a dictionary of matplotlib.Axes objects. Will be None if plotting is not done, or only partially filled if the plotting process ends early.

'writer': pandas.ExcelWriter or None The pandas ExcelWriter used to create the output Excel file; will be None if the output results were not saved to Excel.

4.2 Fitting GUI

The fitting submodule of mcetl provides functions and GUIs for fitting data. mcetl uses the lmfit[22] library for fitting data using non-linear least-squares minimization. For in-depth discussion on the theory and available methods/models for curve fitting, the lmfit[23] documentation is highly recommended.

Please note that the fitting submodule is included in mcetl to allow a fairly basic curve fitting option. If other curve fitting software is available, such as fityk[24] or Origin[25], the author highly recommends their usage instead of mcetl.

4.2.1 Basic Usage

To use the fitting GUI in mcetl, simply do:

```
from mcetl import fitting
fitting.launch_fitting_gui()
```

A window will then appear to select the data file(s) to be fit and the Excel file for saving the results. No other setup is required.

Usage of the fitting GUI is fairly straightforward, and the GUI should notify the user if any issues occur and allow correction without terminating the program.

After doing the fitting, the fit results and plots will be saved to Excel.

Note: Currently, mcetl only provides functions and a GUI for performing peak fitting. A later release of mcetl (v0.5 or v0.6) is slated to add general fitting routines, to allow fitting arbitrary models to data. Further, that release should also add the option to save the fitting options from the GUI to a file so that the options can be reused for fitting other data, similar to how the plotting GUI does it.

---

[22] https://lmfit.github.io/lmfit-py/
[23] https://lmfit.github.io/lmfit-py/
[24] https://fityk.nieto.pl
[25] https://originlab.com

4.2.2 Advanced Usage

To be added.

4.3 Plotting GUI

The plotting module of mcetl provides a GUI for plotting data. mcetl uses the Matplotlib[26] library for plotting, which allows for saving high quality images in several formats.

Please note that the plotting submodule is included in mcetl to allow a fairly basic plotting option. Further, its development will be slower than the main GUI or the fitting submoduule since it requires covering many more options and is not a priority of the author. If other plotting software is available, such as SciDAVis[27], Veusz[28], or Origin[29], the author highly recommends their usage instead of mcetl.

4.3.1 Summary of Features

The plotting GUI has the following features:

- plot data using line and/or scatter plots, and allow customization of the colors, style, and size of the markers/lines.
- specify complex layouts composed of multiple rows and/or columns of axes and inset axes
- add twin axes to plot multiple dependent responses (such as viscosity and density as functions of temperature)
- add secondary axes to plot the same data on different scales, using user-specified conversions (such as plotting temperature in both Fahrenheit and Celsius)
- specify figure dimensions and dots per inch (dpi)
- add formatted annotations (text, arrows, and lines) anywhere on the figure
- allow specifying peak positions, which will optionally place markers and/or labels above each peak position on each dataset.

In addition, changes to Matplotlib's rcParams[30] can be specified, which allows additional control over the style and formatting of the figures.

4.3.2 Basic Usage

To use the plotting GUI in mcetl, simply do:

```
from mcetl import plotting

# add some changes to Matplotlib's rcparams
changes = {
    'font.serif': 'Times New Roman',
    'font.family': 'serif',
    'font.size': 12,
    'xtick.direction': 'in',
```

(continues on next page)

[26] https://matplotlib.org
[27] https://scidavis.sourceforge.net/
[28] https://veusz.github.io
[29] https://originlab.com
[30] https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams

(continued from previous page)

```
    'ytick.direction': 'in'
}
plotting.launch_plotting_gui(mpl_changes=changes)
```

Similar to fitting, a window will then appear to select the data file(s) to be plotted, and no other setup is required for doing plotting.

When plotting, the image of the plots can be saved to all formats supported by Matplotlib[31], including tiff, jpg, png, svg, and pdf. If the Pillow[32] library is installed, additional options are given to allow saving compressed files for some formats, such as tiff, in order to reduce the file size.

4.3.3 Advanced Usage

To be added.

4.3.4 Saving/Reopening Plots

Using the GUI, the layout of the plots can be saved to apply to other figures later, which is referred to in the GUI as saving the Figure Theme. The necessary data will be saved with the file extension ".figjson", which is just a json file.

Further, the data for the plots can be saved to a csv file so that the entire plot can be recreated.

To reopen a figure saved through mcetl, do:

```
plotting.load_previous_figure()
```

which will open a window to select the csv and (optionally) figjson files to use.

4.4 Importing Data

mcetl allows importing data from various sources such as text files, csv files, fixed-width files, and spreadsheet-like files (xlsx, xls, xlsm, xlsb, ods, etc.). Note that the default installation of mcetl only supports importing from spreadsheet-like files with either xlsx or xlsm file extensions (the file types supported by openpyxl[33]). To import data from other spreadsheet-like files, the appropriate Python library must be installed. For example, to read xls files, the xlrd[34] library would need to be installed.

Figure 1 below shows the Data Import window. The meanings of the fields are as follows:

- Columns to import: which columns in the raw data file to import
- Start row: the first row to use from the raw data file
- End row: the last row to use from the raw data file

In addition, if using the Main GUI through *mcetl.launch_main_gui()* (page 67) and processing data, then a section will be added to the bottom of the import window to select the column numbers for each unique variable ('x' and 'y' in Figure 1) for the selected DataSource.

Figure 2 shows examples of different start rows, end rows, data columns, and separators to help show the meanings of these terms. In Python, counting starts at 0, so if you wished to use the first row, for example, you would put 0 as the

[31] https://matplotlib.org
[32] https://python-pillow.org/
[33] https://openpyxl.readthedocs.io/en/stable/
[34] https://github.com/python-excel/xlrd

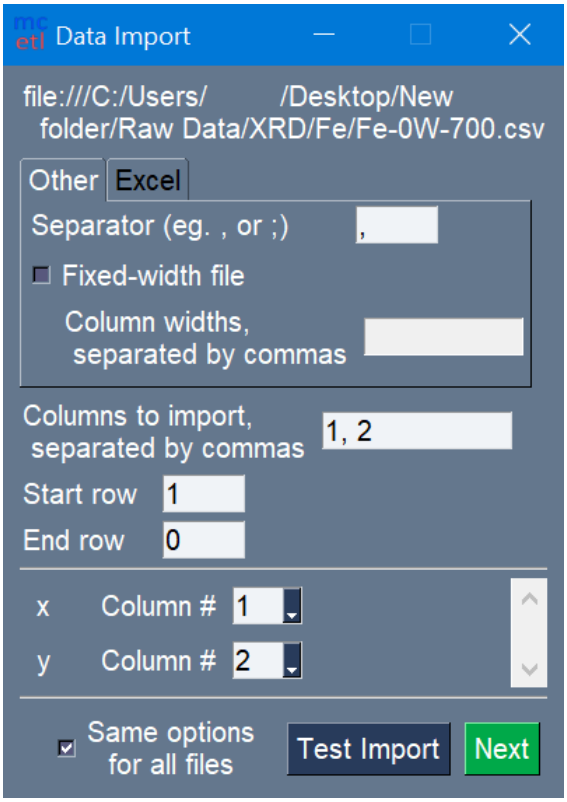

Fig. 1: Figure 1: Options for importing raw data. Columns for unique variables for a DataSource are also selected.

start row. For the end row, it is counting up from the bottom, so the bottom-most row is row 0, the row above it is row 1, etc.

For most file types the only other unique item that needs filled out is the separator. Common separators for files are commas (`,`), semicolons (`;`), tabs (`\t`), and whitespace (`\s+`). Regular expressions[35] can also be used for the separator.

For fixed-width files, the widths of each column also need to be specified, as seen in Figure 3.

For importing data from existing spreadsheet-like files (xlsx, xls, etc.), there are additional options to choose (Figure 4). The assumption when importing data from a spreadsheet-like file is that the data has a common, repeating layout. The options are as follows:

- Sheet to use: which sheet in the file to use
- First Column: the first column in the sheet to use
- Last Column: the last column in the sheet to use
- Number of columns per entry: how many columns are associated with one set of data in the selected sheet; the total number data entries that will be extracted from the file will be equal to (Last Column + 1 – First Column) / Number of columns per entry. For example, if the First Column is column 0, the Last Column is column 11, and there are 4 columns per entry, then the total number of sets of data is (11 + 1 – 0) / 4 = 3.

It is recommended to start from the top of the Data Import window, and work down because various fields are filled out automatically when selecting the sheet to use, number of columns per dataset, etc.

At any time, the "Test Import" button can be pressed to look at how the imported data will look. When the data source

[35] https://docs.python.org/3/howto/regex.html

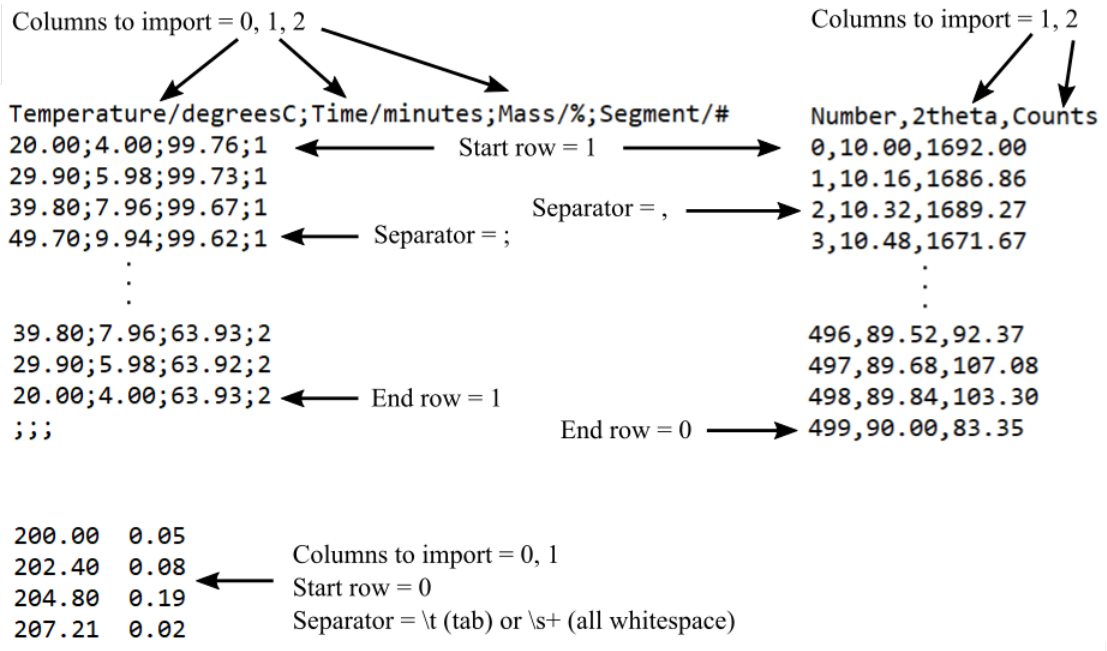

Fig. 2: Figure 2: Three raw data files with different import options.

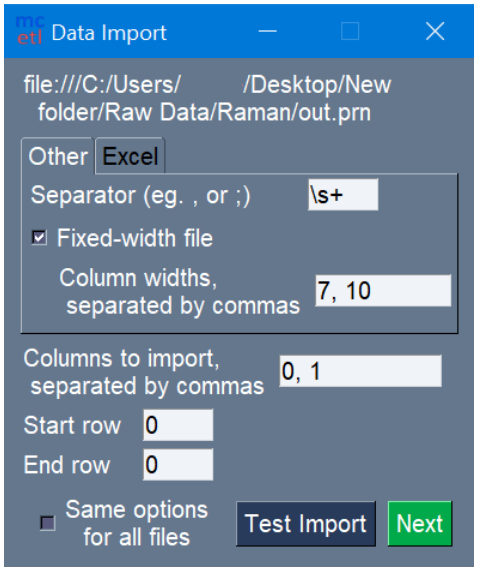

Fig. 3: Figure 3: Importing raw data from fixed-width file.

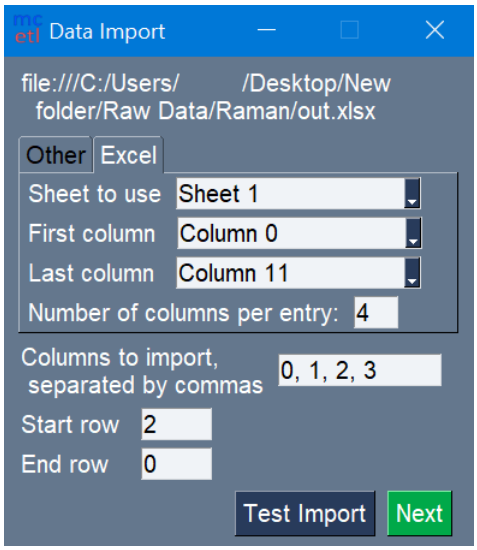

Fig. 4: Figure 4: Importing raw data from a spreadsheet-like file.

is a spreadsheet-like file and the columns per entry is less than the total columns, two windows will show up, with the first window corresponding to the total set of data starting at the first column and ending at the last column, and the second window showing the individual data entries. Any errors that would have occurred during data importing will simply give a pop-up warning window when using "Test Import", rather than causing any issues, so it is best to use "Test Import" if you are unsure if the import options are correct.

4.5 Writing to Excel

mcetl gives the option to write results to Excel when using either *mcetl.launch_main_gui()* (page 67) or *fitting.launch_fitting_gui()* (page 38) from *mcetl.fitting* (page 35). This section covers some information that is good to know when writing to Excel.

4.5.1 Appending to Existing File

mcetl allows appending to existing Excel files, but it should be used with caution, namely for two reasons: high memory usage, and potentially losing Excel objects.

Memory Usage

When appending to an existing Excel file, mcetl loads the Excel workbook using the Python library openpyxl[36]; however, the conversion from Excel to Python can increase the necessary memory by more than an order of magnitude. For example, an Excel file that is 10 MB on disk will require several hundred MB of RAM when opened in Python. Users should to be aware of this fact, so that a MemoryError does not occur when loading existing files. If in doubt, write to a new file using mcetl, and then copy the sheets to the desired file within Excel.

[36] https://openpyxl.readthedocs.io/en/stable/

Losing Excel Objects

openpyxl can convert the majority of objects within an Excel file, such as cell values and styles, defined styles for the workbook, and charts. If the existing Excel file is simply tabulated data and charts, then it should have no issues. However, there are some Excel objects which openpyxl cannot read, such as shapes and inserted equations, so these non-convertable objects will be lost when appending to a file.

If there is any doubt whether openpyxl can read an object within an Excel file, it is a good idea to create a copy of the file before appending to it with mcetl to test whether all objects are transferred, or use mcetl to write to a new file and then copy its contents to the existing file using Excel.

4.5.2 Excel Styles

When writing to Excel, using either *mcetl.launch_main_gui()* (page 67) (using a *DataSource* (page 54) to set the style) or *fitting.launch_fitting_gui()* (page 38), the style of the cells in the Excel workbook can be customized. The *gallery section* (page 85) of the documentation shows two spreadsheets showing the default styles used for both the launch_main_gui function and the launch_fitting_gui function.

There are seven different styles that can be specified, each with an even and odd variant (eg. 'header_even' and 'header_odd'), allowing for fourteen total styles to be specified, making the Excel output visually distinct. The base names for the seven styles are:

- 'header': The style used for the headers.
- 'subheader': The style used for the subheaders.
- 'columns': The style for all of the data columns.
- 'fitting_header': The style used for the headers when writing results from fitting.
- 'fitting_subheader':The style used for the subheaders when writing fit results.
- 'fitting_columns': The style for the data and parameter columns for fit results.
- 'fitting_descriptors': The style for the two columns specifying descriptions about the fit (whether it was successful, the goodness of fit, etc.).

The styles with the 'fitting' prefix are only used if writing results from fitting to Excel.

The styles are specified using a dictionary containing the above base names with '_even' or '_odd' appended (eg. 'header_even') as keys, and either a nested dictionary or an openpyxl `NamedStyle`[37] object as values.

Using Named Styles

To make the styles usable in the output Excel file, there are two options. The first is to use openpyxl `NamedStyle`[38] objects in the dictionary. The second is to use a dictionary, and include the 'name' key in the dictionary to set the NamedStyle's name.

Note: Once a Named Style is added to an Excel workbook, it cannot be overwritten using mcetl. That means that if appending to an existing workbook and trying to set a Named Style to the name of an existing style in the workbook, the format of the existing style will be used rather than the input style. To fix this, just delete or rename any styles that need to be changed within the Excel workbook before writing more data using mcetl.

Some examples of valid inputs that create NamedStyles are shown below:

[37] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle
[38] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle

```
from openpyxl.styles import (
    Alignment, Border, Font, NamedStyle, PatternFill, Side
)

partial_styles = {
    # can use an openpyxl NamedStyle
    'header_even': NamedStyle(
        name='Even Header',
        font=Font(size=12, bold=True),
        fill=PatternFill(fill_type='solid', start_color='F9B381', end_color='F9B381'),
        border=Border(bottom=Side(style='thin')),
        alignment=Alignment(horizontal='center', vertical='center', wrap_text=True),
        number_format='0.00'
    ),
    # or use a dictionary with a 'name' key
    'header_odd': {
        'name': 'Odd Header',
        'font': Font(size=12, bold=True),
        'fill': PatternFill(fill_type='solid', start_color='73A2DB', end_color='73A2DB
↪'),
        'border': Border(bottom=Side(style='thin')),
        'alignment': Alignment(horizontal='center', vertical='center', wrap_
↪text=True),
        'number_format': '0.00'
    },
    # can replace all openpyxl objects with dict to not even need to import openpyxl
    'subheader_odd': {
        'name': 'Odd Subheader',
        'font': dict(size=12, bold=True),
        'fill': dict(fill_type='solid', start_color='73A2DB', end_color='73A2DB'),
        'border': dict(bottom=dict(style='thin')),
        'alignment': dict(horizontal='center', vertical='center', wrap_text=True),
        'number_format': '0.00'
    },
    # can also reference already created NamedStyles
    'subheader_even': 'Odd Subheader'
}
```

Using Anonymous Styles

Anonymous styles will properly format the cells in the output Excel file, but their names will not be available styles in the Excel file. In addition, anonymous styles also have a much faster write time than Named Styles, taking ~ 50% less time to write. So if processing speed is a concern, using anonymous styles is a good choice.

An easy way to create anonymous styles is to first create the NamedStyle, like above, and then replace NamedStyle with dict and remove the 'name' key. Doing so with the styles from the previous section gives:

```
partial_styles = {
    # replace NamedStyle with dict and remove name=''
    'header_even': dict(
        font=Font(size=12, bold=True),
        fill=PatternFill(fill_type='solid', start_color='F9B381', end_color='F9B381'),
        border=Border(bottom=Side(style='thin')),
        alignment=Alignment(horizontal='center', vertical='center', wrap_text=True),
        number_format='0.00'
    ),
```

(continues on next page)

(continued from previous page)

```
    # remove the 'name' key
    'header_odd': {
        'font': Font(size=12, bold=True),
        'fill': PatternFill(fill_type='solid', start_color='73A2DB', end_color='73A2DB
↪'),
        'border': Border(bottom=Side(style='thin')),
        'alignment': Alignment(horizontal='center', vertical='center', wrap_
↪text=True),
        'number_format': '0.00'
    },
    # remove 'name' and replace all openpyxl objects with dict to not even need to␣
↪import openpyxl
    'subheader_odd': {
        'font': dict(size=12, bold=True),
        'fill': dict(fill_type='solid', start_color='73A2DB', end_color='73A2DB'),
        'border': dict(bottom=dict(style='thin')),
        'alignment': dict(horizontal='center', vertical='center', wrap_text=True),
        'number_format': '0.00'
    }
}
```

Using Unformatted Styles

To make a style unformatted (use the Excel default format), simply set its value to either None or an empty dictionary.

```
# both produces same, default style in the output Excel file
partial_styles = {'header_even': {}, 'header_odd': None}
```

To make all styles unformatted, do one of the following dictionary comprehensions:

```
from mcetl import DataSource

unformatted_styles = {style: {} for style in DataSource.excel_styles.keys()}
unformatted_styles2 = {style: None for style in DataSource.excel_styles.keys()}
```

Validate Styles

To ensure that the input style dictionary is valid, DataSource provides the *test_excel_styles()* (page 58) static method, which will indicate the keys of all of the styles are not valid and their error tracebacks:

```
from mcetl import DataSource
from openpyxl.styles import Font, NamedStyle

good_styles = {
    'header_even': NamedStyle(
            name='header_even',
            font=Font(size=12, bold=True),
        ),
    'header_odd': {
        'font': dict(size=12, bold=True),
    }
}
```

(continues on next page)

(continued from previous page)

```
bad_styles = {
    'header_even': dict(
            name='Header',
            font=Font(size=12, bold=True),
            number_format=1 # wrong since number_format must be a string or None
        ),
    'header_odd': {
        'font': dict(size='string'), # wrong since font size must be a float
    }
}

DataSource.test_excel_styles(good_styles) # returns True
DataSource.test_excel_styles(bad_styles) # returns False
```

4.6 Example Data & Programs

mcetl comes with the ability to generate raw data to emulate several different materials characterization techniques. Further, example programs are available to show basic usage of mcetl.

4.6.1 Generating Example Data

Files for example data from characterization techniques can be created using:

```
from mcetl import raw_data
raw_data.generate_raw_data()
```

Data produced by the generate_raw_data function covers the following characterization techniques:

- X-ray diffraction (XRD)
- Fourier-transform infrared spectroscopy (FTIR)
- Raman spectroscopy
- Thermogravimetric analysis (TGA)
- Differential scanning calorimetry (DSC)
- Rheometry
- Uniaxial tensile tests
- Pore size measurements

4.6.2 Example Programs

Example programs[39] are available to show basic usage of mcetl. The examples include:

- Generating raw data
- Using the main GUI
- Using the fitting GUI
- Using the plotting GUI

[39] https://github.com/derb12/mcetl/tree/main/examples

- Reopening a figure saved with the plotting GUI

The example program for using the main GUI contains all necessary inputs for processing the example raw data generated by the generate_raw_data function as described above and is an excellent resource for creating new DataSource objects.

CHAPTER

FIVE

API REFERENCE

Warning: mcetl is going to switch GUI backends in a later version (v0.5 or v0.6), so any api that is not referenced in the Quick Start or Tutorial sections should be used with caution since it is liable to be changed or removed.

5.1 mcetl

mcetl provides user interfaces for processing, fitting, and plotting data.

mcetl is focused on easing the time required to process data files. It does this by allowing the user to define DataSource objects which contains the information for reading files for that DataSource (such as what separator to use, which rows and columns to use, labels for the columns, etc.), the calculations that will be performed on the data, and the options for writing the data to Excel (formatting, placement in the worksheet, etc.).

In addition, mcetl provides fitting and plotting user interfaces that can be used without any prior setup.

Subpackages include:

mcetl.fitting **(page 35)** Contains functions that ease the fitting of data. The main entry is through mcetl.fitting.launch_fitting_gui, but also contains useful functions without needing to launch a GUI.

mcetl.plotting **(page 52)** Contains functions that ease the plotting of data. The main entry is through mcetl.plotting.launch_plotting_gui. Can also reopen a previouly saved figure using mcetl.plotting.load_previous_figure.

@author: Donald Erb Created on Jul 15, 2020

5.1.1 Subpackages

mcetl.fitting

Contains functions that ease the fitting of data. The main entry is through mcetl.fitting.launch_fitting_gui, but also contains useful functions without needing to launch a GUI.

@author: Donald Erb Created on Nov 15, 2020

Submodules

mcetl.fitting.fitting_gui

Provides a GUI to fit data using lmfit Models and save the results to Excel.

@author: Donald Erb Created on May 24, 2020

Notes

openpyxl is imported within fit_to_excel to reduce the import time of the module, and is only imported if saving fit results to Excel.

Module Contents

Classes

ResultsPlot (page 36)	Shows the results of a fit to allow user to decide if fit was acceptable.
SimpleEmbeddedFigure (page 36)	A window containing just an embedded figure and a close button.

Functions

fit_dataframe (page 37)	Creates a GUI to select data from a dataframe for fitting.
fit_to_excel (page 37)	Outputs the relevant data from peak fitting to an Excel file.
launch_fitting_gui (page 38)	Convenience function to fit dataframe(s) and write their results to Excel.

class mcetl.fitting.fitting_gui.**ResultsPlot**(*fit_result*)

Bases: *mcetl.plot_utils.EmbeddedFigure* (page 69)

Shows the results of a fit to allow user to decide if fit was acceptable.

Parameters **fit_result** (*lmfit.ModelResult*) -- The fit result to display.

Initialize self. See help(type(self)) for accurate signature.

event_loop(*self*)

Handles the event loop for the GUI.

Returns Returns True if the Continue button was pressed, otherwise False.

Return type bool[40]

class mcetl.fitting.fitting_gui.**SimpleEmbeddedFigure**(*dataframe*, *gui_values*)

Bases: *mcetl.plot_utils.EmbeddedFigure* (page 69)

A window containing just an embedded figure and a close button.

Parameters

[40] https://docs.python.org/3/library/functions.html#bool

- **dataframe** (*pd.DataFrame*) -- The dataframe that contains the x and y data.
- **gui_values** (*dict*[41]) -- A dictionary of values needed for plotting.

Initialize self. See help(type(self)) for accurate signature.

event_loop(*self*)
Handles the event loop for the GUI.

Notes

This function should typically be overwritten by a subclass, and should typically return any desired values from the embedded figure.

This simple implementation makes the window visible, and closes the window as soon as anything in the window is clicked.

mcetl.fitting.fitting_gui.**fit_dataframe**(*dataframe*, *user_inputs=None*)
Creates a GUI to select data from a dataframe for fitting.

Parameters

- **dataframe** (*pd.DataFrame*) -- A pandas dataframe containing the data to be fit.
- **user_inputs** (*dict*[42]*, optional*) -- Values to use as the default inputs in the GUI.

Returns

- **fit_result** (*lmfit.model.ModelResult or None*) -- A lmfit.ModelResult object, which gives information for the fit done on the dataset. Is None if fitting was skipped.
- **values_df** (*pd.DataFrame or None*) -- The dataframe containing the x and y data, the y data for every individual model, the summed y data of all models, and the background, if present. Is None if fitting was skipped.
- **params_df** (*pd.DataFrame or None*) -- The dataframe containing the value and standard error associated with all of the parameters in the fitting (eg. coefficients for the baseline, areas and sigmas for each peak). Is None if fitting was skipped.
- **descriptors_df** (*pd.DataFrame or None*) -- The dataframe which contains some additional information about the fitting. Currently has the adjusted r squared, reduced chi squared, the Akaike information criteria, the Bayesian information criteria, and the minimization method used for fitting. Is None if fitting was skipped.
- **gui_values** (*dict or None*) -- The values selected in the GUI for all of the various fields, which can be used to reuse the values from a past interation. Is None if fitting was skipped.

mcetl.fitting.fitting_gui.**fit_to_excel**(*values_dataframe*, *params_dataframe*, *descriptors_dataframe*, *excel_writer_handler*, *sheet_name=None*, *plot_excel=False*)
Outputs the relevant data from peak fitting to an Excel file.

Parameters

- **values_dataframe** (*pd.DataFrame*) -- The dataframe containing the x and y data, the y data for every individual model, the summed y data of all models, and the background, if present.

[41] https://docs.python.org/3/library/stdtypes.html#dict
[42] https://docs.python.org/3/library/stdtypes.html#dict

- **params_dataframe** (*pd.DataFrame*) -- The dataframe containing the value and standard error associated with all of the parameters in the fitting (eg. coefficients for the baseline, areas and sigmas for each peak).
- **descriptors_dataframe** (*pd.DataFrame*) -- The dataframe which contains some additional information about the fitting. Currently has the adjusted r squared, reduced chi squared, the Akaike information criteria, the Bayesian information criteria, and the minimization method used for fitting.
- **excel_writer_handler** (mcetl.excel_writer.ExcelWriterHandler (page 59)) -- The ExcelWriterHandler that contains the pandas ExcelWriter object for writing to Excel, and the styles for styling the cells in Excel.
- **sheet_name** (*str*[43], *optional*) -- The Excel sheet name.
- **plot_excel** (*bool*[44], *optional*) -- If True, will create a simple plot in Excel that plots the raw x and y data, the data for each peak, the total of all peaks, and the background if it is present.

mcetl.fitting.fitting_gui.**launch_fitting_gui**(*dataframe=None*, *gui_values=None*, *excel_writer=None*, *save_excel=True*, *plot_excel=True*, *mpl_changes=None*, *save_when_done=True*, *excel_formats=None*)

Convenience function to fit dataframe(s) and write their results to Excel.

Parameters

- **dataframe** (*pd.DataFrame or list/tuple, optional*) -- The dataframe or list/tuple of dataframes to fit.
- **gui_values** (*dict*[45], *optional*) -- A dictionary containing the default gui values to pass to fit_dataframe.
- **excel_writer** (*pd.ExcelWriter, optional*) -- The Excel writer used to save the results to Excel. If input, the engine must be "openpyxl".
- **save_excel** (*bool*[46], *optional*) -- If True (default), then the fit results will be saved to an Excel file.
- **plot_excel** (*bool*[47], *optional*) -- If True (default), then the fit results will be plotted in the Excel file (if saving).
- **mpl_changes** (*dict*[48], *optional*) -- A dictionary of changes to apply to matplotlib's rcParams file, which affects how plots look.
- **save_when_done** (*bool*[49], *optional*) -- If True (default), then the Excel file will be saved once all dataframes are fit.
- **excel_formats** (*dict*[50], *optional*) -- A dictionary of formats to use when writing to Excel. The dictionary must have one of the following keys:

[43] https://docs.python.org/3/library/stdtypes.html#str
[44] https://docs.python.org/3/library/functions.html#bool
[45] https://docs.python.org/3/library/stdtypes.html#dict
[46] https://docs.python.org/3/library/functions.html#bool
[47] https://docs.python.org/3/library/functions.html#bool
[48] https://docs.python.org/3/library/stdtypes.html#dict
[49] https://docs.python.org/3/library/functions.html#bool
[50] https://docs.python.org/3/library/stdtypes.html#dict

'fitting_header_even', 'fitting_header_odd', 'fitting_subheader_even', 'fitting_subheader_odd', 'fitting_columns_even', 'fitting_columns_odd', 'fitting_descriptors_even', 'fitting_descriptors_odd'

The values for each key must be a dictionary, with keys in this internal dictionary representing keyword arguments for openpyxl's NamedStyle or openpyxl.styles.NamedStyle objects. See mcetl.excel_writer.ExcelWriterHandler.styles as an example for the dictionary.

Returns

- **fit_results** (*list(lmfit.models.ModelResult)*) -- A list of lmfit.ModelResult objects, which give information for each of the fits done on the dataframes.
- **gui_values** (*dict, optional*) -- A dictionary containing the default gui values to pass to fit_dataframe.
- **proceed** (*bool*) -- True if the fitting gui was not exited from prematurely, otherwise, the value is False. Useful when calling this function from an outside function that needs to know whether to continue doing peak fitting.

mcetl.fitting.fitting_utils

Provides utility functions, classes, and constants for the fitting module.

Useful functions are put here in order to prevent circular importing within the other files.

@author : Donald Erb Created on Nov 18, 2020

Notes

The functions get_model_name, get_model_object, and get_gui_name are to be used, rather than referring to the constants _TOTAL_MODELS and _GUI_MODELS because the implementation of these constants may change in the future while the output of the functions can be kept constant.

Module Contents

Functions

get_gui_name (page 40)	Returns the name used in GUIs for the input model, eg. 'GaussianModel' -> 'Gaussian'.
get_is_peak (page 40)	Determines if the input model is registered as a peak function.
get_model_name (page 40)	Converts the model name used in GUIs to the model class name.
get_model_object (page 40)	Returns the model object given a model name, eg. 'Gaussian' -> lmfit.models.GaussianModel.
numerical_area (page 41)	Computes the numerical area of a peak using the trapezoidal method.
numerical_extremum (page 41)	Computes the numerical maximum or minimum for a peak.
numerical_fwhm (page 41)	Computes the numerical full-width-at-half-maximum of a peak.

continues on next page

Table 3 – continued from previous page

numerical_mode (page 42)	Computes the numerical mode (x-value at which the extremum occurs) of a peak.
print_available_models (page 42)	Prints out a dictionary of all models supported by mcetl.
r_squared (page 42)	Calculates r^2 and adjusted r^2 for a fit.
r_squared_model_result (page 42)	Calculates r^2 and adjusted r^2 for a fit, given an lmfit.ModelResult.
subtract_linear_background (page 42)	Returns y-values after subtracting a linear background constructed from points.

mcetl.fitting.fitting_utils.**get_gui_name**(*model*)

Returns the name used in GUIs for the input model, eg. 'GaussianModel' -> 'Gaussian'.

Parameters **model** (*str*[51]) -- The input model string.

Returns The model's name as it appears in GUIs.

Return type str[52]

Notes

This is a convenience function to be used so that the internals of how model names are specified can change while code can always use this function.

mcetl.fitting.fitting_utils.**get_is_peak**(*model*)

Determines if the input model is registered as a peak function.

Parameters **model** (*str*[53]) -- The name of the model. Can either be the GUI name (eg. 'Gaussian') or the class name (eg. 'GaussianModel').

Returns **is_peak** -- True if the input model is within _TOTAL_MODELS and _TOTAL_MODELS[model]['is_peak'] is True. If the model cannot be found, or if _TOTAL_MODELS[model]['is_peak'] is False, then returns False.

Return type bool[54]

mcetl.fitting.fitting_utils.**get_model_name**(*model*)

Converts the model name used in GUIs to the model class name.

For example, converts 'Gaussian' to 'GaussianModel' and 'Skewed Gaussian' to 'SkewedGaussianModel'.

Useful so that code can always refer to the original model name and not be affected if the name of the model used in GUIs changes. Also ensures that user-input model names are correctly interpreted.

Parameters **model** (*str*[55]) -- The model name used within a GUI or input by a user.

Returns **output** -- The class name of the model, which can give other information by using _TOTAL_MODELS[output].

Return type str[56]

Raises **KeyError:** -- Raised if the input model name is not valid.

mcetl.fitting.fitting_utils.**get_model_object**(*model*)

Returns the model object given a model name, eg. 'Gaussian' -> lmfit.models.GaussianModel.

[51] https://docs.python.org/3/library/stdtypes.html#str
[52] https://docs.python.org/3/library/stdtypes.html#str
[53] https://docs.python.org/3/library/stdtypes.html#str
[54] https://docs.python.org/3/library/functions.html#bool
[55] https://docs.python.org/3/library/stdtypes.html#str
[56] https://docs.python.org/3/library/stdtypes.html#str

Parameters **model** (*str*[57]) -- The name of the model desired. Can either be the model class name, such as 'GaussianModel', or the name used in GUIs, such as 'Gaussian.

Returns **output** -- The class corresponding to the input model name.

Return type lmfit.Model

mcetl.fitting.fitting_utils.**numerical_area**(*x*, *y*)

Computes the numerical area of a peak using the trapezoidal method.

Parameters

- **x** (*array-like*) -- The x-values for the peak model.
- **y** (*array-like*) -- The y-values for the peak model.

Returns The integrated area of the peak, using the trapezoidal method.

Return type float[58]

mcetl.fitting.fitting_utils.**numerical_extremum**(*y*)

Computes the numerical maximum or minimum for a peak.

Parameters **y** (*array-like*) -- The y-values for the peak model.

Returns **extremum** -- The extremum with the highest absolute value from the input y data. Assumes the peak is negative if abs(min(y)) > abs(max(y)).

Return type float[59]

Notes

Use np.nanmin and np.nanmax instead of min and max in order to convert the output to a numpy dtype. This way, all of the numerical calculations using the output of this function will work, even if y is a list or tuple.

mcetl.fitting.fitting_utils.**numerical_fwhm**(*x*, *y*)

Computes the numerical full-width-at-half-maximum of a peak.

Parameters

- **x** (*array-like*) -- The x-values for the peak model.
- **y** (*array-like*) -- The y-values for the peak model.

Returns The calculated full-width-at-half-max of the peak. If there are not at least two x-values at which y = extremum_y / 2, then None is returned.

Return type float[60] or None[61]

[57] https://docs.python.org/3/library/stdtypes.html#str
[58] https://docs.python.org/3/library/functions.html#float
[59] https://docs.python.org/3/library/functions.html#float
[60] https://docs.python.org/3/library/functions.html#float
[61] https://docs.python.org/3/library/constants.html#None

Notes

First finds the x-values where y - extremum_y / 2 changes signs, and then uses linear interpolation to approximate the x-values at which y - extremum_y / 2 = 0

mcetl.fitting.fitting_utils.**numerical_mode**(*x*, *y*)
Computes the numerical mode (x-value at which the extremum occurs) of a peak.

Parameters

- **x** (*array-like*) -- The x-values for the peak model.
- **y** (*array-like*) -- The y-values for the peak model.

Returns The x-value at which the extremum in y occurs.

Return type float[62]

mcetl.fitting.fitting_utils.**print_available_models**()
Prints out a dictionary of all models supported by mcetl.

Also prints out details for each model, including its class, the name used when displaying in GUIs, its parameters, and whether it is considered a peak function.

mcetl.fitting.fitting_utils.**r_squared**(*y_data*, *y_fit*, *num_variables=1*)
Calculates r^2 and adjusted r^2 for a fit.

Parameters

- **y_data** (*array-like*) -- The experimental y data.
- **y_fit** (*array-like*) -- The calculated y from fitting.
- **num_variables** (*int*[63]*, optional*) -- The number of variables used by the fitting model.

Returns

- **r_sq** (*float*) -- The r squared value for the fitting.
- **r_sq_adj** (*float*) -- The adjusted r squared value for the fitting, which takes into account the number of variables in the fitting model.

mcetl.fitting.fitting_utils.**r_squared_model_result**(*fit_result*)
Calculates r^2 and adjusted r^2 for a fit, given an lmfit.ModelResult.

Parameters **fit_result** (*lmfit.ModelResult*) -- The ModelResult object from a fit.

Returns The r^2 and adjusted r^2 values for the fitting.

Return type tuple[64](float[65], float[66])

mcetl.fitting.fitting_utils.**subtract_linear_background**(*x*, *y*, *background_points*)
Returns y-values after subtracting a linear background constructed from points.

Parameters

- **x** (*array-like*) -- The x-values of the data.
- **y** (*array-like*) -- The y-values of the data.

[62] https://docs.python.org/3/library/functions.html#float
[63] https://docs.python.org/3/library/functions.html#int
[64] https://docs.python.org/3/library/stdtypes.html#tuple
[65] https://docs.python.org/3/library/functions.html#float
[66] https://docs.python.org/3/library/functions.html#float

- **background_points** (*list*[67] *(list*[68] *(float*[69]*, float*[70]*))*) -- A list containing the [x, y] values for each point representing the background.

Returns **y_subtracted** -- The input y-values, after subtracting the background.

Return type np.ndarray

Notes

Assumes the background is represented by lines connecting each of the specified background points.

mcetl.fitting.models

Provides additional models for fitting data.

@author : Donald Erb Created on Nov 29, 2020

Module Contents

Classes

BreitWignerFanoModel (page 43)	A modified version of lmfit's BreitWignerModel.

Functions

breit_wigner_fano_alt (page 43)	An alternate Breit-Wigner-Fano lineshape that uses height rather than amplitude.

class mcetl.fitting.models.**BreitWignerFanoModel**(*independent_vars=['x']*, *prefix=''*, *nan_policy='raise'*, ***kwargs*)

Bases: lmfit.model.Model[71]

A modified version of lmfit's BreitWignerModel.

Initializes with q=-3 rather than q=1, which gives a better peak. Also defines terms for fwhm, height, x_mode, and maximum that were not included in lmfit's implementation, where x_mode is the x position at the maximum y-value, and maximum is the maximum y-value. Further, the lmfit implementation uses amplitude as value for bwf as abs(x) approaches infinity, which is different than for most other peaks where amplitude is defined as the peak area.

Compared to lmfit's implementation, height = amplitude * q**2, and sigma (this model) = sigma (lmfit) / 2.

guess(*self*, *data*, *x=None*, *negative=False*, ***kwargs*)
Estimate initial model parameter values from data.

mcetl.fitting.models.**breit_wigner_fano_alt**(*x*, *height=1.0*, *center=0.0*, *sigma=1.0*, *q=-3.0*)

[67] https://docs.python.org/3/library/stdtypes.html#list
[68] https://docs.python.org/3/library/stdtypes.html#list
[69] https://docs.python.org/3/library/functions.html#float
[70] https://docs.python.org/3/library/functions.html#float
[71] https://lmfit.github.io/lmfit-py/model.html#lmfit.model.Model

An alternate Breit-Wigner-Fano lineshape that uses height rather than amplitude.

breit_wigner_fano(x, height, center, sigma, q) = height * (1 + (x - center) / (q * sigma))**2 / (1 + ((x - center) / sigma)**2)

mcetl.fitting.peak_fitting

Functions for creating and fitting a model with peaks and a background and plotting the results.

Also contains two classes that create windows to allow selection of peak positions and background points.

@author: Donald Erb Created on Sep 14, 2019

Module Contents

Classes

BackgroundSelector (page 44)	A window for selecting points to define the background of data.
PeakSelector (page 45)	A window for selecting peaks on a plot, along with peak width and type.

Functions

find_peak_centers (page 46)	Creates a list containing the peaks found and peaks accepted.
fit_peaks (page 47)	Takes x,y data, finds the peaks, fits the peaks, and returns all relevant information.
plot_confidence_intervals (page 49)	Plot the data, the fit, and the fit +- n_sig * sigma confidence intervals.
plot_fit_results (page 50)	Plot the raw data, best fit, and residuals.
plot_individual_peaks (page 50)	Plots each individual peak in the composite model and the total model.
plot_peaks_for_model (page 51)	Plot the peaks found or added, as well as the ones found but rejected from the fitting.

class mcetl.fitting.peak_fitting.**BackgroundSelector**(*x*, *y*, *click_list=None*)

Bases: *mcetl.plot_utils.EmbeddedFigure* (page 69)

A window for selecting points to define the background of data.

Parameters

- **x** (*array-like*) -- The x-values to be plotted.
- **y** (*array-like*) -- The y-values to be plotted.
- **click_list** (*list*[72] *(list*[73] *(float*[74]*, float*[75]*)), optional*) -- A list of se-

[72] https://docs.python.org/3/library/stdtypes.html#list
[73] https://docs.python.org/3/library/stdtypes.html#list
[74] https://docs.python.org/3/library/functions.html#float
[75] https://docs.python.org/3/library/functions.html#float

lected points (lists of x, y values) on the plot.

axis_2
The secondary axis on the figure which has no events.

Type plt.Axes

Initialize self. See help(type(self)) for accurate signature.

event_loop(*self*)
Handles the event loop for the GUI.

Returns A list of the [x, y] values for the selected points on the figure.

Return type list[76](list[77](float[78], float[79]))

class mcetl.fitting.peak_fitting.**PeakSelector**(*x*, *y*, *click_list=None*, *initial_peak_width=1*, *subtract_background=False*, *background_type='PolynomialModel'*, *background_kwargs=None*, *bkg_min=-np.inf*, *bkg_max=np.inf*, *default_model=None*)
Bases: *mcetl.plot_utils.EmbeddedFigure* (page 69)

A window for selecting peaks on a plot, along with peak width and type.

Parameters

- **x** (*array-like*) -- The x-values to be plotted and fitted.
- **y** (*array-like*) -- The y-values to be plotted and fitted.
- **click_list** (*list[80], optional*) -- A nested list, with each entry corresponding to a peak. Each entry has the following layout: [[lmfit model, sigma function, aplitude function], [peak center, peak height, peak width]] where lmfit model is something like 'GaussianModel'. The first entry in the list comes directly from the mcetl.peak_fitting.peak_transformer function.
- **initial_peak_width** (*int[81] or float[82]*) -- The initial peak width input in the plot.
- **subtract_background** (*bool[83]*) -- If True, will subtract the background before showing the plot.
- **background_type** (*str[84]*) -- String corresponding to a model in lmfit.models; used to fit the background.
- **background_kwargs** (*dict[85]*) -- Any keyword arguments needed to initialize the background model
- **bkg_min** (*float[86] or int[87]*) -- Minimum x value to use for initially fitting the background.

[76] https://docs.python.org/3/library/stdtypes.html#list
[77] https://docs.python.org/3/library/stdtypes.html#list
[78] https://docs.python.org/3/library/functions.html#float
[79] https://docs.python.org/3/library/functions.html#float
[80] https://docs.python.org/3/library/stdtypes.html#list
[81] https://docs.python.org/3/library/functions.html#int
[82] https://docs.python.org/3/library/functions.html#float
[83] https://docs.python.org/3/library/functions.html#bool
[84] https://docs.python.org/3/library/stdtypes.html#str
[85] https://docs.python.org/3/library/stdtypes.html#dict
[86] https://docs.python.org/3/library/functions.html#float
[87] https://docs.python.org/3/library/functions.html#int

- **bkg_max** (*float*[88] *or int*[89]) -- Maximum x value to use for initially fitting the background.
- **default_model** (*str*[90]) -- The initial model to have selected on the plot, corresponds to a model in lmfit.models.

background
The y-values for the background function.

Type array-like

Initialize self. See help(type(self)) for accurate signature.

event_loop(*self*)
Handles the event loop for the GUI.

Returns A nested list, with each entry corresponding to a peak. Each entry has the following layout: [model, peak center, peak height, peak width] where model is something like 'Gaussian'. The first entry in the list comes directly from the mcetl.peak_fitting.peak_transformer function.

Return type list[91]

mcetl.fitting.peak_fitting.**find_peak_centers**(*x*, *y*, *additional_peaks=None*, *height=None*, *prominence=np.inf*, *x_min=-np.inf*, *x_max=np.inf*)
Creates a list containing the peaks found and peaks accepted.

Parameters

- **y** (*x,*) -- x and y values for the fitting.
- **additional_peaks** (*list*[92]*, optional*) -- Peak centers that are input by the user and automatically accepted as peaks if within x_min and x_max.
- **height** (*int*[93] *or float*[94]*, optional*) -- Height that a peak must have for it to be considered a peak by scipy's find_peaks.
- **prominence** (*int*[95] *or float*[96]) -- Prominence that a peak must have for it to be considered a peak by scipy's find_peaks.
- **x_min** (*int*[97] *or float*[98]) -- Minimum x value used in fitting the data.
- **x_max** (*int*[99] *or float*[100]) -- Maximum x value used in fitting the data.

Returns

- **peaks_found** (*list*) -- A list of all the peak centers found.
- **peaks_accepted** (*list*) -- A list of peak centers that were accepted because they were within x_min and x_max.

[88] https://docs.python.org/3/library/functions.html#float
[89] https://docs.python.org/3/library/functions.html#int
[90] https://docs.python.org/3/library/stdtypes.html#str
[91] https://docs.python.org/3/library/stdtypes.html#list
[92] https://docs.python.org/3/library/stdtypes.html#list
[93] https://docs.python.org/3/library/functions.html#int
[94] https://docs.python.org/3/library/functions.html#float
[95] https://docs.python.org/3/library/functions.html#int
[96] https://docs.python.org/3/library/functions.html#float
[97] https://docs.python.org/3/library/functions.html#int
[98] https://docs.python.org/3/library/functions.html#float
[99] https://docs.python.org/3/library/functions.html#int
[100] https://docs.python.org/3/library/functions.html#float

Notes

Uses scipy's signal.find_peaks to find peaks matching the specifications, and adds those peaks to a list of additionally specified peaks.

mcetl.fitting.peak_fitting.**fit_peaks**(*x*, *y*, *height=None*, *prominence=np.inf*, *center_offset=1.0*, *peak_width=1.0*, *default_model='PseudoVoigtModel'*, *subtract_background=False*, *bkg_min=-np.inf*, *bkg_max=np.inf*, *min_sigma=0.0*, *max_sigma=np.inf*, *min_method='least_squares'*, *x_min=-np.inf*, *x_max=np.inf*, *additional_peaks=None*, *model_list=None*, *background_type='PolynomialModel'*, *background_kwargs=None*, *fit_kws=None*, *vary_voigt=False*, *fit_residuals=False*, *num_resid_fits=5*, *min_resid=0.05*, *debug=False*, *peak_heights=None*)

Takes x,y data, finds the peaks, fits the peaks, and returns all relevant information.

Parameters

- **y** (*x,*) -- x and y values for the fitting.
- **height** (*int*[101] *or float*[102]) -- Height that a peak must have for it to be considered a peak by scipy's find_peaks.
- **prominence** (*int*[103] *or float*[104]) -- Prominence that a peak must have for it to be considered a peak by scipy's find_peaks.
- **center_offset** (*int*[105] *or float*[106]) -- Value that determines the min and max parameters for the center of each peak. min = center - center_offset, max = center + center_offset.
- **peak_width** (*int*[107] *or float*[108]) -- A guess at the full-width-half-max of the peaks. When first estimating the peak parameters, only the data from x - peak_width / 2 to x + peak_width / 2 is used to fit the peak.
- **default_model** (*str*[109]) -- Model used if the model_list is None or does not have enough values for the number of peaks found. Must correspond to one of the built-in models in lmfit.models.
- **subtract_background** (*bool*[110]) -- If True, it will fit the background with the model in background_type.
- **bkg_min** (*int*[111] *or float*[112]) -- Minimum x value to use for initially fitting the background.

[101] https://docs.python.org/3/library/functions.html#int
[102] https://docs.python.org/3/library/functions.html#float
[103] https://docs.python.org/3/library/functions.html#int
[104] https://docs.python.org/3/library/functions.html#float
[105] https://docs.python.org/3/library/functions.html#int
[106] https://docs.python.org/3/library/functions.html#float
[107] https://docs.python.org/3/library/functions.html#int
[108] https://docs.python.org/3/library/functions.html#float
[109] https://docs.python.org/3/library/stdtypes.html#str
[110] https://docs.python.org/3/library/functions.html#bool
[111] https://docs.python.org/3/library/functions.html#int
[112] https://docs.python.org/3/library/functions.html#float

- **bkg_max** (*int*[113] *or float*[114]) -- Maximum x value to use for initially fitting the background.
- **min_sigma** (*int*[115] *or float*[116]) -- Minimum value for the sigma for each peak; typically better to not set to any value other than 0.
- **max_sigma** (*int*[117] *or float*[118]) -- maximum value for the sigma for each peak; typically better to not set to any value other than infinity.
- **min_method** (*str*[119]) -- Minimization method used for fitting.
- **x_min** (*int*[120] *or float*[121]) -- Minimum x value used in fitting the data.
- **x_max** (*int*[122] *or float*[123]) -- Maximum x value used in fitting the data.
- **additional_peaks** (*list*[124]) -- Peak centers that are input by the user and automatically accepted as peaks.
- **model_list** (*list*[125]) -- List of strings, with each string corresponding to one of the models in lmfit.models.
- **background_type** (*str*[126]) -- String corresponding to a model in lmfit.models; used to fit the background.
- **background_kwargs** (*dict*[127]*, optional*) -- Any keyword arguments needed to initialize the background model.
- **fit_kws** (*dict*[128]) -- Keywords to be passed on to the minimizer.
- **vary_voigt** (*bool*[129]) -- If True, will allow the gamma parameter in the Voigt model to be varied as an additional variable; if False, gamma will be set equal to sigma.
- **fit_residuals** (*bool*[130]) -- If True, it will attempt to fit the residuals after the first fitting to find hidden peaks, add them to the model, and fit the data again.
- **num_resid_fits** (*int*[131]) -- Maximum number of times the program will loop to fit the residuals.
- **min_resid** (*int*[132] *or float*[133]) -- used as the prominence when finding peaks in the residuals, in which the prominence is set to min_resid * (y_max - y_min).
- **debug** (*bool*[134]) -- If True, will create plots at various portions of the code showing the peak initialization, initial fits and backgrounds, and residuals.

[113] https://docs.python.org/3/library/functions.html#int
[114] https://docs.python.org/3/library/functions.html#float
[115] https://docs.python.org/3/library/functions.html#int
[116] https://docs.python.org/3/library/functions.html#float
[117] https://docs.python.org/3/library/functions.html#int
[118] https://docs.python.org/3/library/functions.html#float
[119] https://docs.python.org/3/library/stdtypes.html#str
[120] https://docs.python.org/3/library/functions.html#int
[121] https://docs.python.org/3/library/functions.html#float
[122] https://docs.python.org/3/library/functions.html#int
[123] https://docs.python.org/3/library/functions.html#float
[124] https://docs.python.org/3/library/stdtypes.html#list
[125] https://docs.python.org/3/library/stdtypes.html#list
[126] https://docs.python.org/3/library/stdtypes.html#str
[127] https://docs.python.org/3/library/stdtypes.html#dict
[128] https://docs.python.org/3/library/stdtypes.html#dict
[129] https://docs.python.org/3/library/functions.html#bool
[130] https://docs.python.org/3/library/functions.html#bool
[131] https://docs.python.org/3/library/functions.html#int
[132] https://docs.python.org/3/library/functions.html#int
[133] https://docs.python.org/3/library/functions.html#float
[134] https://docs.python.org/3/library/functions.html#bool

- **peak_heights** (*list*[135]) -- A list of peak heights.

Returns

output -- A dictionary of lists. For most lists, each entry corresponds to the results of a peak fitting. The dictionary has the following keys (in this order if unpacked into a list):

'resid_peaks_found': All peak centers found during fitting of the residuals. Each list entry corresponds to a separate fitting.

'resid_peaks_accepted': Peak centers found during fitting of the residuals which were accepted as true peak centers. Each list entry corresponds to a separate fitting.

'peaks_found': All peak centers found during the initial peak fitting of the original data.

'peaks_accepted': Peaks centers found during the initial peak fitting of the original data which were accepted as true peak centers

'initial_fits': List of initial fits. Each list entry corresponds to a separate fitting.

'fit_results': List of lmfit's ModelResult objects, which contain the majority of information needed. Each list entry corresponds to a separate fitting.

'individual_peaks': Nested list of y-data values for each peak. Each list entry corresponds to a separate fitting.

'best_values: Nested list of best fit parameters (such as amplitude, fwhm, etc.) for each fitting. Each list entry corresponds to a separate fitting.

Return type dict[136]

Notes

Uses several of the functions within this module to directly take (x,y) data and do peak fitting. All relevant data is returned from this function, so it has quite a dense input and output, but it is well worth it!

For the minimization method (min_method), the 'least_squares' method gives fast performance compared to most other methods while still having good convergence. The 'leastsq' is also very good, and seems less susceptible to getting caught in local minima compared to 'least_squares', but takes longer to evaluate. A best practice could be to use 'least_squares' during initial testing and then using 'leastsq' for final calculations.

Global optimizers do not seem to work on complicated models (or rather I cannot get them to work), so using the local optimizers 'least_squares' or 'leastsq' and adjusting the inputs will have to suffice.

Most relevant data is contained within output['fit_results'], such as the parameters for each peak, as well as all the error associated with the fitting.

Within this function: params == parameters for all of the peaks bkg_params == parameters for the background composite_params == parameters for peaks + background

model == CompositeModel object for all of the peaks background == Model object for the background composite_model == CompositeModel object for the peaks + background

peaks_accepted are not necessarily the peak centers after fitting, just the peak centers that were found. These values can be used for understanding the peak selection process.

mcetl.fitting.peak_fitting.**plot_confidence_intervals**(*fit_result*, *n_sig=3*, *return_figure=False*)

Plot the data, the fit, and the fit +- n_sig * sigma confidence intervals.

[135] https://docs.python.org/3/library/stdtypes.html#list
[136] https://docs.python.org/3/library/stdtypes.html#dict

Parameters

- **fit_result** (*lmfit.ModelResult*) -- The ModelResult object for the fitting.
- **n_sig** (*float*[137], *optional*) -- The multiple of the standard error to use as the plotted error. The default is 3.
- **return_figure** (*bool*[138], *optional*) -- If True, will return the created figure; otherwise, it will display the figure using plt.show(block=False) (default).

Returns If return_figure is True, the figure is returned; otherwise, None is returned.

Return type plt.Figure or None[139]

Notes

This function assumes that the independant variable in the fit_result was labeled 'x', as is standard for all built-in lmfit models.

mcetl.fitting.peak_fitting.**plot_fit_results**(*fit_result*, *label_rsq=False*, *plot_initial=False*, *return_figure=False*)

Plot the raw data, best fit, and residuals.

Parameters

- **fit_result** (*lmfit.ModelResult or list*[140]*(lmfit.ModelResult)*) -- A ModelResult object from lmfit; can be a list of ModelResults, in which case, the initial fit will use fit_result[0], and the best fit will use fit_result[-1].
- **label_rsq** (*bool*[141], *optional*) -- If True, will put a label with the adjusted r squared value of the fitting.
- **plot_initial** (*bool*[142], *optional*) -- If True, will plot the initial fitting as well as the best fit.
- **return_figure** (*bool*[143], *optional*) -- If True, will return the created figure; otherwise, it will display the figure using plt.show(block=False) (default).

Returns If return_figure is True, the figure is returned; otherwise, None is returned.

Return type plt.Figure or None[144]

Notes

This function assumes that the independant variable in the fit_result was labeled 'x', as is standard for all built-in lmfit models.

mcetl.fitting.peak_fitting.**plot_individual_peaks**(*fit_result*, *individual_peaks*, *background_subtracted=False*, *plot_subtract_background=False*, *plot_separate_background=False*, *plot_w_background=False*, *return_figures=False*)

[137] https://docs.python.org/3/library/functions.html#float
[138] https://docs.python.org/3/library/functions.html#bool
[139] https://docs.python.org/3/library/constants.html#None
[140] https://docs.python.org/3/library/stdtypes.html#list
[141] https://docs.python.org/3/library/functions.html#bool
[142] https://docs.python.org/3/library/functions.html#bool
[143] https://docs.python.org/3/library/functions.html#bool
[144] https://docs.python.org/3/library/constants.html#None

Plots each individual peak in the composite model and the total model.

Parameters

- **fit_result** (*lmfit.ModelResult*) -- The ModelResult object from the fitting.
- **individual_peaks** (*dict*[145]) -- A dictionary with keys corresponding to the model prefixes in the fitting, and their values corresponding to their y values.
- **background_subtracted** (*bool*[146]) -- Whether or not the background was subtracted in the fitting.
- **plot_subtract_background** (*bool*[147]) -- If True, subtracts the background from the raw data as well as all the peaks, so everything is nearly flat.
- **plot_separate_background** (*bool*[148]) -- If True, subtracts the background from just the individual peaks, while also showing the raw data with the composite model and background.
- **plot_w_background** (*bool*[149]) -- If True, has the background added to all peaks, i.e. it shows exactly how the composite model fits the raw data.
- **return_figures** (*bool*[150]*, optional*) -- If True, will return the created figures; otherwise, it will display the figures using plt.show(block=False) (default).

Returns If return_figures is True, the list of created figures is returned. Otherwise, None is returned.

Return type list[151](plt.Figure) or None[152]

Notes

This function assumes that the independant variable in the fit_result was labeled 'x', as is standard for all built-in lmfit models.

mcetl.fitting.peak_fitting.**plot_peaks_for_model**(*x*, *y*, *x_min*, *x_max*, *peaks_found*, *peaks_accepted*, *additional_peaks*)

Plot the peaks found or added, as well as the ones found but rejected from the fitting.

Parameters

- **x** (*array-like*) -- x data used in the fitting.
- **y** (*array-like*) -- y data used in the fitting.
- **x_min** (*float*[153] *or int*[154]) -- Minimum x values used for the fitting procedure.
- **x_max** (*float*[155] *or int*[156]) -- Maximum x values used for the fitting procedure.
- **peaks_found** (*list*[157]) -- A list of x values corresponding to all peaks found throughout the peak fitting and peak finding process, as well as those input by the user.

[145] https://docs.python.org/3/library/stdtypes.html#dict
[146] https://docs.python.org/3/library/functions.html#bool
[147] https://docs.python.org/3/library/functions.html#bool
[148] https://docs.python.org/3/library/functions.html#bool
[149] https://docs.python.org/3/library/functions.html#bool
[150] https://docs.python.org/3/library/functions.html#bool
[151] https://docs.python.org/3/library/stdtypes.html#list
[152] https://docs.python.org/3/library/constants.html#None
[153] https://docs.python.org/3/library/functions.html#float
[154] https://docs.python.org/3/library/functions.html#int
[155] https://docs.python.org/3/library/functions.html#float
[156] https://docs.python.org/3/library/functions.html#int
[157] https://docs.python.org/3/library/stdtypes.html#list

- **peaks_accepted** (*list*[158]) -- A list of x values corresponding to peaks found throughout the peak fitting and peak finding process that were accepted into the model.
- **additional_peaks** (*list*[159]) -- A list of peak centers that were input by the user.

mcetl.plotting

Contains functions that ease the plotting of data. The main entry is through mcetl.plotting.launch_plotting_gui. Can also reopen a previouly saved figure using mcetl.plotting.load_previous_figure.

@author: Donald Erb Created on Nov 15, 2020

Submodules

mcetl.plotting.plotting_gui

GUIs to plot data using various plot layouts and save the resulting figures.

@author: Donald Erb Created on Jun 28, 2020

mcetl.plotting.plotting_gui.**COLORS**
A tuple with values that are used in GUIs to select the color to plot with in matplotlib. The default is ('None', 'Black', 'Blue', 'Red', Green', 'Chocolate', 'Magenta', 'Cyan', 'Orange', 'Coral', 'Dodgerblue').

Type tuple[160](str[161], ..)

mcetl.plotting.plotting_gui.**LINE_MAPPING**
A dictionary with keys that are displayed in GUIs, and values that are used by matplotlib to specify the line style. The default is {

'None': '', 'Solid': '-', 'Dashed': '--', 'Dash-Dot': '-.', 'Dot': ':', 'Dash-Dot-Dot': (0, [0.75 * plt.rcParams['lines.dashdot_pattern'][0]] +

plt.rcParams['lines.dashdot_pattern'][1:] + plt.rcParams['lines.dashdot_pattern'][-2:])

}

Type dict[162]

mcetl.plotting.plotting_gui.**MARKERS**
A tuple of strings for the default markers to use for plotting. The default is (' None', 'o Circle', 's Square', '^ Triangle-Up', 'D Diamond', 'v Triangle-Down', 'p Pentagon', '< Triangle-Left', '> Triangle-Right', '* Star').

Type tuple[163](str[164], ..)

mcetl.plotting.plotting_gui.**TIGHT_LAYOUT_PAD**
The padding placed between the edge of the figure and the edge of the canvas; used by matplotlib's tight_layout option. Default is 0.3.

Type float[165]

[158] https://docs.python.org/3/library/stdtypes.html#list
[159] https://docs.python.org/3/library/stdtypes.html#list
[160] https://docs.python.org/3/library/stdtypes.html#tuple
[161] https://docs.python.org/3/library/stdtypes.html#str
[162] https://docs.python.org/3/library/stdtypes.html#dict
[163] https://docs.python.org/3/library/stdtypes.html#tuple
[164] https://docs.python.org/3/library/stdtypes.html#str
[165] https://docs.python.org/3/library/functions.html#float

mcetl.plotting.plotting_gui.**TIGHT_LAYOUT_H_PAD**
The height (vertical) padding between axes in a figure; used by matplotlib's tight_layout option. Default is 0.6.

Type float[166]

mcetl.plotting.plotting_gui.**TIGHT_LAYOUT_W_PAD**
The width (horizontal) padding between axes in a figure; used by matplotlib's tight_layout option. Default is 0.6.

Type float[167]

Module Contents

Functions

launch_plotting_gui (page 53)	Convenience function to plot lists of dataframes with matplotlib.
load_previous_figure (page 53)	Load the options and the data to recreate a figure.

mcetl.plotting.plotting_gui.**launch_plotting_gui**(*dataframes=None*, *mpl_changes=None*, *input_fig_kwargs=None*, *input_axes=None*, *input_values=None*)
Convenience function to plot lists of dataframes with matplotlib.

Wraps the plotting in a context manager that applies the changes to the matplotlib rcParams.

Parameters

- **dataframes** (*list*[168] *(list*[169] *(pd.DataFrame))*) -- A nested list of lists of pandas DataFrames. Each list of DataFrames will create one figure.
- **mpl_changes** (*dict*[170]) -- Changes to matplotlib's rcParams file to alter the figure.
- **input_fig_kwargs** (*dict*[171]*, optional*) -- The fig_kwargs from a previous session. Only used if reloading a figure.
- **input_axes** (*dict*[172]*, optional*) -- A dictionary of plt.Axes objects from a reloaded session.
- **input_values** (*dict*[173]*, optional*) -- The values needed to recreate the previous gui window from a reloaded figure.

Returns **figures** -- A nested list of lists, with each entry containing the matplotlib Figure, and a dictionary containing the Axes.

Return type list[174](list[175](plt.Figure, dict[176](str[177], plt.Axes)))

[166] https://docs.python.org/3/library/functions.html#float
[167] https://docs.python.org/3/library/functions.html#float
[168] https://docs.python.org/3/library/stdtypes.html#list
[169] https://docs.python.org/3/library/stdtypes.html#list
[170] https://docs.python.org/3/library/stdtypes.html#dict
[171] https://docs.python.org/3/library/stdtypes.html#dict
[172] https://docs.python.org/3/library/stdtypes.html#dict
[173] https://docs.python.org/3/library/stdtypes.html#dict
[174] https://docs.python.org/3/library/stdtypes.html#list
[175] https://docs.python.org/3/library/stdtypes.html#list
[176] https://docs.python.org/3/library/stdtypes.html#dict
[177] https://docs.python.org/3/library/stdtypes.html#str

mcetl.plotting.plotting_gui.**load_previous_figure**(*filename=None*, *new_rc_changes=None*)

Load the options and the data to recreate a figure.

Parameters

- **filename** (*str*[178], *optional*) -- The filepath string to the csv data file to be opened.
- **new_rc_changes** (*dict*[179], *optional*) -- New changes to matplotlib's rcParams file to alter the saved figure.

Returns **figures** -- A list of figures (with len=1) using the loaded data. If no file is selected, then figures = None.

Return type list[180] or None[181]

Notes

Will load the data from the csv file specified by filename. If there also exists a .figjson file with the same name as the csv file, it will be loaded to set the figure layout. Otherwise, a new figure is created.

5.1.2 Submodules

mcetl.data_source

The DataSource class contains all needed information for importing, processing, and saving data.

@author: Donald Erb Created on Jul 31, 2020

Module Contents

Classes

DataSource (page 54)	Used to give default settings for importing data and various functions based on the source.

class mcetl.data_source.**DataSource**(*name*, *, *functions=None*, *column_labels=None*, *column_numbers=None*, *start_row=0*, *end_row=0*, *separator=None*, *file_type=None*, *num_files=1*, *unique_variables=None*, *unique_variable_indices=None*, *xy_plot_indices=None*, *figure_rcparams=None*, *excel_writer_styles=None*, *excel_row_offset=0*, *excel_column_offset=0*, *entry_separation=0*, *sample_separation=0*, *label_entries=True*)

Used to give default settings for importing data and various functions based on the source.

Parameters

- **name** (*str*[182]) -- The name of the DataSource. Used when displaying the DataSource in a GUI.

[178] https://docs.python.org/3/library/stdtypes.html#str
[179] https://docs.python.org/3/library/stdtypes.html#dict
[180] https://docs.python.org/3/library/stdtypes.html#list
[181] https://docs.python.org/3/library/constants.html#None
[182] https://docs.python.org/3/library/stdtypes.html#str

- **functions** (*list*[183] *or tuple*[184]*, optional*) -- A list or tuple of various Function objects (*CalculationFunction* (page 64) or *PreprocessFunction* (page 65) or *SummaryFunction* (page 65)) that will be used to process data for the DataSource. The order the Functions are performed in is as follows: PreprocessFunctions, CalculationFunctions, SummaryFunctions, with functions of the same type being performed in the same order as input.
- **column_labels** (*tuple*[185]*(str*[186]*) or list*[187]*(str*[188]*), optional*) -- A list/tuple of strings that will be used to label columns in the Excel output, and to label the pandas DataFrame columns for the data.
- **column_numbers** (*tuple*[189]*(int*[190]*) or list*[191]*(int*[192]*), optional*) -- The indices of the columns to import from raw data files.
- **start_row** (*int*[193]*, optional*) -- The first row of data to use when importing from raw data files.
- **end_row** (*int*[194]*, optional*) -- The last row of data to use when importing from raw data files. Counts up from the last row, so the last row is 0, the second to last row is 1, etc.
- **separator** (*str*[195]*, optional*) -- The separator or delimeter to use to separate data columns when importing from raw data files. For example, ',' for csv files.
- **file_type** (*str*[196]*, optional*) -- The file extension associated with the data files for the DataSource. For example, 'txt' or 'csv'.
- **num_files** (*int*[197]*, optional*) -- The number of data files per sample for the DataSource. Only used when using keyword search for files.
- **unique_variables** (*list*[198]*(str*[199]*) or tuple*[200]*(str*[201]*), optional*) -- The names of all columns from the imported raw data that are needed for calculations. For example, if importing thermogravimetric analysis (TGA) data, the unique_variables could be ['temperature', 'mass'].
- **unique_variable_indices** (*list*[202]*(int*[203]*) or tuple*[204]*(int*[205]*), optional*) -- The indices of the columns within column_numbers that correspond with each of the input unique_variables.

[183] https://docs.python.org/3/library/stdtypes.html#list
[184] https://docs.python.org/3/library/stdtypes.html#tuple
[185] https://docs.python.org/3/library/stdtypes.html#tuple
[186] https://docs.python.org/3/library/stdtypes.html#str
[187] https://docs.python.org/3/library/stdtypes.html#list
[188] https://docs.python.org/3/library/stdtypes.html#str
[189] https://docs.python.org/3/library/stdtypes.html#tuple
[190] https://docs.python.org/3/library/functions.html#int
[191] https://docs.python.org/3/library/stdtypes.html#list
[192] https://docs.python.org/3/library/functions.html#int
[193] https://docs.python.org/3/library/functions.html#int
[194] https://docs.python.org/3/library/functions.html#int
[195] https://docs.python.org/3/library/stdtypes.html#str
[196] https://docs.python.org/3/library/stdtypes.html#str
[197] https://docs.python.org/3/library/functions.html#int
[198] https://docs.python.org/3/library/stdtypes.html#list
[199] https://docs.python.org/3/library/stdtypes.html#str
[200] https://docs.python.org/3/library/stdtypes.html#tuple
[201] https://docs.python.org/3/library/stdtypes.html#str
[202] https://docs.python.org/3/library/stdtypes.html#list
[203] https://docs.python.org/3/library/functions.html#int
[204] https://docs.python.org/3/library/stdtypes.html#tuple
[205] https://docs.python.org/3/library/functions.html#int

- **xy_plot_indices** (*list*[206] *(int*[207]*, int*[208]*) or tuple*[209] *(int*[210]*, int*[211]*), optional*) -- The indices of the columns after processing that will be the default columns for plotting in Excel.
- **figure_rcparams** (*dict*[212]*, optional*) -- A dictionary containing any changes to Matplotlib's rcParams to use if fitting or plotting.
- **excel_writer_styles** (*dict*[213] *(str*[214]*, None*[215] *or dict*[216] *or str*[217] *or openpyxl.styles.named_styles.NamedStyle*[218]*), optional*) -- A dictionary of styles used to format the output Excel workbook. The following keys are used when writing data from files to Excel:

  'header_even', 'header_odd', 'subheader_even', 'subheader_odd', 'columns_even', 'columns_odd'

  The following keys are used when writing data fit results to Excel:
  'fitting_header_even', 'fitting_header_odd', 'fitting_subheader_even', 'fitting_subheader_odd', 'fitting_columns_even', 'fitting_columns_odd', 'fitting_descriptors_even', 'fitting_descriptors_odd'

  The values for the dictionaries must be either dictionaries, with keys corresponding to keyword inputs for openpyxl's NamedStyle, or NamedStyle objects.
- **excel_row_offset** (*int*[219]*, optional*) -- The first row to use when writing to Excel. A value of 0 would start at row 1 in Excel, 1 would start at row 2, etc.
- **excel_column_offset** (*int*[220]*, optional*) -- The first column to use when writing to Excel. A value of 0 would start at column 'A' in Excel, 1 would start at column 'B', etc.
- **entry_separation** (*int*[221]*, optional*) -- The number of blank columns to insert between data entries when writing to Excel.
- **sample_separation** (*int*[222]*, optional*) -- The number of blank columns to insert between samples when writing to Excel.
- **label_entries** (*bool*[223]*, optional*) -- If True, will add a number to the column labels for each entry in a sample if there is more than one entry. For example, the column label 'data' would become 'data, 1', 'data, 2', etc.

excel_styles
A nested dictionary of dictionaries, used to create openpyxl NamedStyle objects to format the output Excel file.

[206] https://docs.python.org/3/library/stdtypes.html#list
[207] https://docs.python.org/3/library/functions.html#int
[208] https://docs.python.org/3/library/functions.html#int
[209] https://docs.python.org/3/library/stdtypes.html#tuple
[210] https://docs.python.org/3/library/functions.html#int
[211] https://docs.python.org/3/library/functions.html#int
[212] https://docs.python.org/3/library/stdtypes.html#dict
[213] https://docs.python.org/3/library/stdtypes.html#dict
[214] https://docs.python.org/3/library/stdtypes.html#str
[215] https://docs.python.org/3/library/constants.html#None
[216] https://docs.python.org/3/library/stdtypes.html#dict
[217] https://docs.python.org/3/library/stdtypes.html#str
[218] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle
[219] https://docs.python.org/3/library/functions.html#int
[220] https://docs.python.org/3/library/functions.html#int
[221] https://docs.python.org/3/library/functions.html#int
[222] https://docs.python.org/3/library/functions.html#int
[223] https://docs.python.org/3/library/functions.html#bool

Type dict[224](dict[225])

lengths
A list of lists of lists of integers, corresponding to the number of columns in each individual entry in the total dataframes for the DataSource. Used to split the concatted dataframe back into individual dataframes for each dataset.

Type list[226]

references
A list of dictionaries, with each dictionary containing the column numbers for each unique variable and calculation for the merged dataframe of each dataset.

Type list[227]

Raises

- **ValueError**[228] -- Raised if the input name is a blank string, or if either excel_row_offset or excel_column_offset is < 0.
- **TypeError**[229] -- Raised if one of the input functions is not a valid mcetl.FunctionBase object.
- **IndexError**[230] -- Raised if the number of data columns is less than the number of unique variables.

merge_datasets(*self*, *dataframes*)
Merges all entries and samples into one dataframe for each dataset.

Also sets the length attribute, which will later be used to separate each dataframes back into individual dataframes for each entry.

Parameters **dataframes** (*list*[231] *(list*[232] *(list*[233] *(pd.DataFrame)))*) -- A nested list of list of lists of dataframes.

Returns **merged_dataframes** -- A list of dataframes.

Return type list[234](pd.DataFrame)

print_column_labels_template(*self*)
Convenience function that will print a template for all the column headers.

Column headers account for all of the columns imported from raw data, the columns added by CalculationFunctions, and the columns added by SummaryFunctions.

Returns **label_template** -- The list of strings that serves as a template for the necessary input for column_labels for the DataSource.

Return type list[235](str[236])

[224] https://docs.python.org/3/library/stdtypes.html#dict
[225] https://docs.python.org/3/library/stdtypes.html#dict
[226] https://docs.python.org/3/library/stdtypes.html#list
[227] https://docs.python.org/3/library/stdtypes.html#list
[228] https://docs.python.org/3/library/exceptions.html#ValueError
[229] https://docs.python.org/3/library/exceptions.html#TypeError
[230] https://docs.python.org/3/library/exceptions.html#IndexError
[231] https://docs.python.org/3/library/stdtypes.html#list
[232] https://docs.python.org/3/library/stdtypes.html#list
[233] https://docs.python.org/3/library/stdtypes.html#list
[234] https://docs.python.org/3/library/stdtypes.html#list
[235] https://docs.python.org/3/library/stdtypes.html#list
[236] https://docs.python.org/3/library/stdtypes.html#str

split_into_entries (*self*, *merged_dataframes*)
Splits the merged dataset dataframes back into dataframes for each entry.

Parameters **merged_dataframes** (*list*[237] *(pd.DataFrame)*) -- A list of dataframes. Each dataframe will be split into lists of lists of dataframes.

Returns **split_dataframes** -- A list of lists of lists of dataframes, corresponding to entries and samples within each dataset.

Return type list[238](list[239](list[240](pd.DataFrame)))

static test_excel_styles (*styles*)
Tests whether the input styles create valid Excel styles with openpyxl.

Parameters **styles** (*dict*[241] *(str*[242]*, None*[243] *or dict*[244] *or str*[245] *or openpyxl.styles.named_styles.NamedStyle*[246]*)*) -- The dictionary of styles to test. Values in the dictionary can either be None, a nested dictionary with the necessary keys and values to create an openpyxl NamedStyle, a string (which would refer to another NamedStyle.name), or openpyxl.styles.NamedStyle objects.

Returns Returns True if all input styles successfully create openpyxl NamedStyle objects; otherwise, returns False.

Return type bool[247]

Notes

This is just a wrapper of *ExcelWriterHandler.test_excel_styles()* (page 61), and is included because DataSource is the main-facing object of mcetl and will be used more often.

mcetl.excel_writer

ExcelWriterHandler class, used to safely open and close files and apply styles.

@author : Donald Erb Created on Dec 9, 2020

Notes

openpyxl is imported within methods of ExcelWriterHandler in order to reduce the import time of mcetl since the writer is only needed if writing to Excel.

[237] https://docs.python.org/3/library/stdtypes.html#list
[238] https://docs.python.org/3/library/stdtypes.html#list
[239] https://docs.python.org/3/library/stdtypes.html#list
[240] https://docs.python.org/3/library/stdtypes.html#list
[241] https://docs.python.org/3/library/stdtypes.html#dict
[242] https://docs.python.org/3/library/stdtypes.html#str
[243] https://docs.python.org/3/library/constants.html#None
[244] https://docs.python.org/3/library/stdtypes.html#dict
[245] https://docs.python.org/3/library/stdtypes.html#str
[246] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle
[247] https://docs.python.org/3/library/functions.html#bool

Module Contents

Classes

ExcelWriterHandler (page 59)	A helper for pandas's ExcelWriter for opening/saving files and applying styles.

class mcetl.excel_writer.**ExcelWriterHandler**(*file_name=None*, *new_file=False*, *styles=None*, *writer=None*, ***kwargs*)

A helper for pandas's ExcelWriter for opening/saving files and applying styles.

This class is used for ensuring that an existing file is closed before saving and/or writing, if appending, and that all desired styles are ready for usage before writing to Excel. Styles can either be openpyxl NamedStyle objects in order to make the style available in the Excel workbook, or dictionaries detailing the style, such as {'font': Font(...), 'border': Border(...), ...}.

Parameters

- **file_name** (*str*[248] *or Path*) -- The file name or path for the Excel file to be created.
- **new_file** (*bool*[249]*, optional*) -- If False (default), will append to an existing file. If True, or if no file currently exists with the given file_name, a new file will be created, even if a file with the same name currently exists.
- **styles** (*dict*[250]*(str*[251]*, None*[252] *or dict*[253] *or str*[254] *or openpyxl.styles.named_styles.NamedStyle*[255]*)*) -- A dictionary of either nested dictionaries used to create openpyxl style objects (Alignment, Font, etc.), a string indicating the name of an openpyxl NamedStyle to use, a NamedStyle object or None (will use the default style if None). All styles in the dictionary will be added to the Excel workbook if they do not currently exist in the workbook. See Examples below to see various valid inputs for styles.
- **writer** (*pd.ExcelWriter or None*[256]) -- The ExcelWriter (_OpenpyxlWriter from pandas) used for writing to Excel. If it is a pd.ExcelWriter, its engine must be "openpyxl".
- ****kwargs** -- Any additional keyword arguments to pass to pd.ExcelWriter.

styles

A nested dictionary of dictionaries, used to create openpyxl NamedStyle objects to include in self.writer.book. The styles are used as a class attribute to ensure that the necessary styles are always included in the Excel book.

Type dict[257](str[258], dict[259])

style_cache

The currently implemented styles within the Excel workbook. Used to quickly apply styles to cells without having to constanly set all of the cell attributes (cell.font, cell.fill, etc.). The dictionary value is a tuple of

[248] https://docs.python.org/3/library/stdtypes.html#str
[249] https://docs.python.org/3/library/functions.html#bool
[250] https://docs.python.org/3/library/stdtypes.html#dict
[251] https://docs.python.org/3/library/stdtypes.html#str
[252] https://docs.python.org/3/library/constants.html#None
[253] https://docs.python.org/3/library/stdtypes.html#dict
[254] https://docs.python.org/3/library/stdtypes.html#str
[255] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle
[256] https://docs.python.org/3/library/constants.html#None
[257] https://docs.python.org/3/library/stdtypes.html#dict
[258] https://docs.python.org/3/library/stdtypes.html#str
[259] https://docs.python.org/3/library/stdtypes.html#dict

the cell attribute name and the value to set. Will either be ('style', string indicating NamedStyle.name) if using NamedStyles or ('_style', openpyxl StyleArray) to indicate an anonomous style. The call to set the cell attribute for a desired key would be setattr(cell, *style_cache[key]).

Type dict[260](str[261], tuple[262](str[263], str[264] or openpyxl.styles.cell_style.StyleArray[265]))

writer
The ExcelWriter (_OpenpyxlWriter from pandas) used for writing to Excel.

Type pd.ExcelWriter

Notes

Either file_name or writer must be specified at initialization.

Examples

Below is a partial example of various allowable input styles. Can be openpyxl NamedStyle, str, None, or dictionary. (Note that NamedStyle, Font, Border, and Side are gotten by importing from openpyxl.styles)

```
>>> styles = {
        # Would make the style 'Even Header' available in the output Excel file
        'fitting_header_even': NamedStyle(
            name='Even Header',
            font=Font(size=12, bold=True),
            border=Border(bottom=Side(style='thin')),
            number_format='0.0'
        ),
        # Would use same format as 'fitting_header_even'
        'fitting_header_odd': 'Even Header',
        # Basically just replaces NamedStyle from 'fitting_header_even' with
        # dict and removes the 'name' key. A new style would not be created
        # in the output Excel file.
        'fitting_subheader_even': dict(
            font=Font(size=12, bold=True),
            aligment=Aligment(bottom=Side(style='thin')),
            number_format='0.0'
        ),
        # Same as 'fitting_subheader_even', but doesn't require importing
        # from openpyxl. Basically just replaces all openpyxl objects with dict.
        'fitting_subheader_odd': dict(
            font=dict(size=12, bold=True),
            aligment=dict(bottom=dict(style='thin')),
            number_format='0.0'
        ),
        # Same as 'fitting_subheader_odd', but will create a NamedStyle (and
        # add the style to the Excel file) since 'name' is within the dictionary.
        'fitting_columns_even': dict(
            name='New Style',
            font=dict(size=12, bold=True),
```

(continues on next page)

[260] https://docs.python.org/3/library/stdtypes.html#dict
[261] https://docs.python.org/3/library/stdtypes.html#str
[262] https://docs.python.org/3/library/stdtypes.html#tuple
[263] https://docs.python.org/3/library/stdtypes.html#str
[264] https://docs.python.org/3/library/stdtypes.html#str
[265] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.cell_style.html#openpyxl.styles.cell_style.StyleArray

(continued from previous page)

```
            aligment=dict(bottom=dict(style='thin')),
            number_format='0.0'
        ),
        # Would use default style set by openpyxl
        'fitting_columns_odd': {},
        # Would also use default style
        'fitting_descriptors_even': None
    }
```

Raises

- **TypeError**[266] -- Raised if both file_name and writer are None.
- **ValueError**[267] -- Raised if the engine of the input writer is not "openpyxl".

add_styles(*self*, *styles*)

Adds styles to the Excel workbook and update self.style_cache.

Parameters **styles** (*dict*[268] *(str*[269]*, None*[270] *or dict*[271] *or str*[272] *or openpyxl.styles.named_styles.NamedStyle*[273]*)*) -- A dictionary of either nested dictionaries used to create openpyxl style objects (Alignment, Font, etc.), a string indicating the name of an openpyxl NamedStyle to use, a NamedStyle object or None (will use the default style if None). All styles in the dictionary will be added to the Excel workbook if they do not currently exist in the workbook.

Notes

The ordering of items within styles will be preserved, so that if two NamedStyles are input with the same name, the one appearing first in the dictionary will be created, and the second will be made to refer to the first.

save_excel_file(*self*)

Tries to save the Excel file, and handles any PermissionErrors.

Saving can be cancelled if other changes to self.writer are desired before saving, or if saving is no longer desired (the file must be open while trying to save to allow cancelling the save).

classmethod test_excel_styles(*cls*, *styles*)

Tests whether the input styles create valid Excel styles using openpyxl.

Parameters **styles** (*dict*[274] *(str*[275]*, None*[276] *or dict*[277] *or str*[278] *or openpyxl.styles.named_styles.NamedStyle*[279]*)*) -- The dictionary of styles to test. Values in the dictionary can either be None, a nested dictionary with the

[266] https://docs.python.org/3/library/exceptions.html#TypeError
[267] https://docs.python.org/3/library/exceptions.html#ValueError
[268] https://docs.python.org/3/library/stdtypes.html#dict
[269] https://docs.python.org/3/library/stdtypes.html#str
[270] https://docs.python.org/3/library/constants.html#None
[271] https://docs.python.org/3/library/stdtypes.html#dict
[272] https://docs.python.org/3/library/stdtypes.html#str
[273] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle
[274] https://docs.python.org/3/library/stdtypes.html#dict
[275] https://docs.python.org/3/library/stdtypes.html#str
[276] https://docs.python.org/3/library/constants.html#None
[277] https://docs.python.org/3/library/stdtypes.html#dict
[278] https://docs.python.org/3/library/stdtypes.html#str
[279] https://openpyxl.readthedocs.io/en/stable/api/openpyxl.styles.named_styles.html#openpyxl.styles.named_styles.NamedStyle

necessary keys and values to create an openpyxl NamedStyle, a string (which would refer to another NamedStyle.name), or openpyxl.styles.NamedStyle objects.

Returns **success** -- Returns True if all input styles successfully create openpyxl NamedStyle objects; otherwise, returns False.

Return type bool[280]

mcetl.file_organizer

Provides GUIs to find files containing combinations of keywords and move files.

@author: Donald Erb Created on Sep 2, 2019

Module Contents

Functions

file_finder (page 62)	Finds files that match the given keywords and file type using a GUI.
file_mover (page 62)	Takes in a list of file paths and moves a copy of each file to the new folder.
manual_file_finder (page 63)	Allows manual selection for the files for the selected samples and datasets.

mcetl.file_organizer.**file_finder**(*file_directory=None*, *file_type=None*, *num_files=None*)

Finds files that match the given keywords and file type using a GUI.

Parameters

- **file_directory** (*str*[281]) -- String for the topmost folder under which all files are searched.
- **file_type** (*str*[282]) -- The file extension that is being searched, eg. csv, txt, pdf.
- **num_files** (*int*[283]) -- The default maximum and minimum number of files to be associated with each search term.

Returns **output_list** -- A nested list of lists containing the file locations as strings for the files that matched the search term. len(output_list) is equal to the number of datasets, len(output_list[i]) is equal to the number of unique keywords for dataset i, and len(outupt_list[i][j]) is equal to the number of files for dataset i and unique keyword j.

Return type list[284]

mcetl.file_organizer.**file_mover**(*file_list*, *new_folder=None*, *skip_same_files=True*)

Takes in a list of file paths and moves a copy of each file to the new folder.

Parameters

[280] https://docs.python.org/3/library/functions.html#bool
[281] https://docs.python.org/3/library/stdtypes.html#str
[282] https://docs.python.org/3/library/stdtypes.html#str
[283] https://docs.python.org/3/library/functions.html#int
[284] https://docs.python.org/3/library/stdtypes.html#list

- **file_list** (*list*[285], *tuple*[286], *or str*[287]) -- A list of strings corresponding to file paths, all of which will have their copies moved.
- **new_folder** (*str*[288] *or Path*) -- The folder to move all of copies of the files in the file_list into.
- **skip_same_files** (*bool*[289]) -- If True, will not move any copied files if they already exist in the destination folder; if False, will rename the copied file and move it to the destination folder.

Returns **new_folder** -- The string of the destination folder location.

Return type str[290]

mcetl.file_organizer.**manual_file_finder**(*file_type=None*)

Allows manual selection for the files for the selected samples and datasets.

Parameters **file_type** (*str*[291], *optional*) -- The desired file extension for all files.

Returns **files** -- A list of lists of lists of file paths. Each list of lists corresponds to a dataset, and each internal list corresponds to a sample in the dataset.

Return type list[292](list[293](list[294](str[295])))

mcetl.functions

Contains the classes for Functions objects.

There are three main types of Functions:

1) **PreprocessFunction: preprocesses the imported data entry; for example, can** separate into multiple data entries or remove data columns.
2) **CalculationFunction: performs a calculation on each of the entries within** each sample within each dataset.
3) **SummaryFunction: performs a calculation once per sample or once per** dataset.

@author: Donald Erb Created on Jul 31, 2020

[285] https://docs.python.org/3/library/stdtypes.html#list
[286] https://docs.python.org/3/library/stdtypes.html#tuple
[287] https://docs.python.org/3/library/stdtypes.html#str
[288] https://docs.python.org/3/library/stdtypes.html#str
[289] https://docs.python.org/3/library/functions.html#bool
[290] https://docs.python.org/3/library/stdtypes.html#str
[291] https://docs.python.org/3/library/stdtypes.html#str
[292] https://docs.python.org/3/library/stdtypes.html#list
[293] https://docs.python.org/3/library/stdtypes.html#list
[294] https://docs.python.org/3/library/stdtypes.html#list
[295] https://docs.python.org/3/library/stdtypes.html#str

Module Contents

Classes

CalculationFunction (page 64)	Function that performs a calculation for every entry in each sample.
PreprocessFunction (page 65)	Function for processing data before performing any calculations.
SummaryFunction (page 65)	Calculation that is only performed once per sample or once per dataset.

class mcetl.functions.**CalculationFunction**(*name*, *target_columns*, *functions*, *added_columns*, *function_kwargs=None*)

Bases: mcetl.functions._FunctionBase

Function that performs a calculation for every entry in each sample.

Parameters

- **name** (*str*[296]) -- The string representation for this object.
- **target_columns** (*str*[297] *or list*[298] *(str*[299]*)* *or tuple*[300] *(str*[301]*)*) -- A string or list/tuple of strings designating the target columns for this object.
- **functions** (*Callable or list*[302] *(Callable, Callable) or tuple*[303] *(Callable, Callable)*) -- The functions that this object uses to process data. If only one function is given, it is assumed that the same function is used for both calculations for the data to be written to Excel and the data to be used in python. If a list/tuple of functions are given, it is assumed that the first function is used for processing the data to write to Excel, and the second function is used for processing the data to be used in python. The function should take args of list(list(pd.DataFrame)), target indices (a list of lists of lists of numbers, corresponding to the column index in each dataframe for each of the target columns), added columns (a list of lists of numbers, corresponding to the column index in the dataframe for each added column), the Excel columns (a list of column names corresponding to the column in Excel for each added column, eg 'A', 'B', if doing the Excel calculations, otherwise None), the first row (the row number of the first row of data in Excel, eg 3). The function should output a list of lists of pd.DataFrames. The Excel columns and first row values are meant to ease the writing of formulas for Excel.
- **added_columns** (*int*[304] *or str*[305] *or list*[306] *(str*[307]*)* *or tuple*[308] *(str*[309]*)*) -- The columns that will be acted upon by this object's functions. If the input is an integer, then it denotes that the functions act on columns that need to be added, with the number of columns affected by the functions being equal to the input

[296] https://docs.python.org/3/library/stdtypes.html#str
[297] https://docs.python.org/3/library/stdtypes.html#str
[298] https://docs.python.org/3/library/stdtypes.html#list
[299] https://docs.python.org/3/library/stdtypes.html#str
[300] https://docs.python.org/3/library/stdtypes.html#tuple
[301] https://docs.python.org/3/library/stdtypes.html#str
[302] https://docs.python.org/3/library/stdtypes.html#list
[303] https://docs.python.org/3/library/stdtypes.html#tuple
[304] https://docs.python.org/3/library/functions.html#int
[305] https://docs.python.org/3/library/stdtypes.html#str
[306] https://docs.python.org/3/library/stdtypes.html#list
[307] https://docs.python.org/3/library/stdtypes.html#str
[308] https://docs.python.org/3/library/stdtypes.html#tuple
[309] https://docs.python.org/3/library/stdtypes.html#str

integer. If the input is a string or list/tuple of strings, it denotes that the functions will change the contents of an existing column(s), whose column names are the inputs.

- **function_kwargs** (*dict*[310] *or list*[311]*(dict*[312]*, dict*[313]*), optional*) -- A dictionary or a list of two dictionaries containing keyword arguments to be passed to the functions. If a list of two dictionaries is given, the first and second dictionaries will be the keyword arguments to pass to the function for processing the data to write to Excel and the function for processing the data to be used in python, respectively. If a single dictionary is given, then it is used for both functions. The default is None, which passes an empty dictionary to both functions.

Raises **ValueError**[314] -- Raised if there is an issue with added_columns or target_columns, or if any key in the input function_kwargs is within self._forbidden_keys.

class mcetl.functions.**PreprocessFunction**(*name*, *target_columns*, *function*, *function_kwargs=None*, *deleted_columns=None*)

Bases: mcetl.functions._FunctionBase

Function for processing data before performing any calculations.

For example, can separate a single data entry into multiple entries depending on a criteria or delete unneeded columns.

Parameters

- **name** (*str*[315]) -- The string representation for this object.
- **target_columns** (*str*[316] *or list*[317]*(str*[318]*) or tuple*[319]*(str*[320]*)*) -- A string or list/tuple of strings designating the target columns for this object.
- **function** (*Callable*) -- The function that this object uses to process data. The function should take args of dataframe and target indices (a list of numbers, corresponding to the column index in the dataframe for each of the target columns), and should return a list of dataframes.
- **function_kwargs** (*dict*[321]*, optional*) -- A dictionary of keywords and values to be passed to the function. The default is None.
- **deleted_columns** (*str*[322] *or list*[323]*(str*[324]*) or tuple*[325]*(str*[326]*), optional*) -- The names of columns that will be deleted by this object's function.

Raises **ValueError**[327] -- Raised if there is an issue with the input name or target_columns.

class mcetl.functions.**SummaryFunction**(*name*, *target_columns*, *functions*, *added_columns*, *function_kwargs=None*, *sample_summary=True*)

Bases: *mcetl.functions.CalculationFunction* (page 64)

[310] https://docs.python.org/3/library/stdtypes.html#dict
[311] https://docs.python.org/3/library/stdtypes.html#list
[312] https://docs.python.org/3/library/stdtypes.html#dict
[313] https://docs.python.org/3/library/stdtypes.html#dict
[314] https://docs.python.org/3/library/exceptions.html#ValueError
[315] https://docs.python.org/3/library/stdtypes.html#str
[316] https://docs.python.org/3/library/stdtypes.html#str
[317] https://docs.python.org/3/library/stdtypes.html#list
[318] https://docs.python.org/3/library/stdtypes.html#str
[319] https://docs.python.org/3/library/stdtypes.html#tuple
[320] https://docs.python.org/3/library/stdtypes.html#str
[321] https://docs.python.org/3/library/stdtypes.html#dict
[322] https://docs.python.org/3/library/stdtypes.html#str
[323] https://docs.python.org/3/library/stdtypes.html#list
[324] https://docs.python.org/3/library/stdtypes.html#str
[325] https://docs.python.org/3/library/stdtypes.html#tuple
[326] https://docs.python.org/3/library/stdtypes.html#str
[327] https://docs.python.org/3/library/exceptions.html#ValueError

Calculation that is only performed once per sample or once per dataset.

Parameters

- **name** (*str*[328]) -- The string representation for this object.
- **target_columns** (*str*[329] *or list*[330]*(str*[331]*) or tuple*[332]*(str*[333]*)*) -- A string or list/tuple of strings designating the target columns for this object.
- **functions** (*Callable or list*[334]*(Callable, Callable) or tuple*[335]*(Callable, Callable)*) -- The functions that this object uses to process data. If only one function is given, it is assumed that the same function is used for both calculations for the data to be written to Excel and the data to be used in python. If a list/tuple of functions are given, it is assumed that the first function is used for processing the data to write to Excel, and the second function is used for processing the data to be used in python. The function should take args of list(list(pd.DataFrame)), target indices (a list of lists of lists of numbers, corresponding to the column index in each dataframe for each of the target columns), added columns (a list of lists of numbers, corresponding to the column index in the dataframe for each added column), the Excel columns (a list of column names corresponding to the column in Excel for each added column, eg 'A', 'B', if doing the Excel calculations, otherwise None), the first row (the row number of the first row of data in Excel, eg 3). The function should output a list of lists of pd.DataFrames. The Excel columns and first row values are meant to ease the writing of formulas for Excel.
- **added_columns** (*int*[336] *or str*[337] *or list*[338]*(str*[339]*) or tuple*[340]*(str*[341]*)*) -- The columns that will be acted upon by this object's functions. If the input is an integer, then it denotes that the functions act on columns that need to be added, with the number of columns affected by the functions being equal to the input integer. If the input is a string or list/tuple of strings, it denotes that the functions will change the contents of an existing column(s), whose column names are the inputs. Further, SummaryFunctions can only modify other SummaryFunction columns with matching sample_summary attributes.
- **function_kwargs** (*dict*[342] *or list*[343]*(dict*[344]*, dict*[345]*), optional*) -- A dictionary or a list of two dictionaries containing keyword arguments to be passed to the functions. If a list of two dictionaries is given, the first and second dictionaries will be the keyword arguments to pass to the function for processing the data to write to Excel and the function for processing the data to be used in python, respectively. The default is None, which passes an empty dictionary to both functions.
- **sample_summary** (*bool*[346]*, optional*) -- If True (default), denotes that the Summa-

[328] https://docs.python.org/3/library/stdtypes.html#str
[329] https://docs.python.org/3/library/stdtypes.html#str
[330] https://docs.python.org/3/library/stdtypes.html#list
[331] https://docs.python.org/3/library/stdtypes.html#str
[332] https://docs.python.org/3/library/stdtypes.html#tuple
[333] https://docs.python.org/3/library/stdtypes.html#str
[334] https://docs.python.org/3/library/stdtypes.html#list
[335] https://docs.python.org/3/library/stdtypes.html#tuple
[336] https://docs.python.org/3/library/functions.html#int
[337] https://docs.python.org/3/library/stdtypes.html#str
[338] https://docs.python.org/3/library/stdtypes.html#list
[339] https://docs.python.org/3/library/stdtypes.html#str
[340] https://docs.python.org/3/library/stdtypes.html#tuple
[341] https://docs.python.org/3/library/stdtypes.html#str
[342] https://docs.python.org/3/library/stdtypes.html#dict
[343] https://docs.python.org/3/library/stdtypes.html#list
[344] https://docs.python.org/3/library/stdtypes.html#dict
[345] https://docs.python.org/3/library/stdtypes.html#dict
[346] https://docs.python.org/3/library/functions.html#bool

ryFunction summarizes a sample; if False, denotes that the SummaryFunction summarizes a dataset.

Raises `ValueError`[347] -- Raised if there is an issue with added_columns or target_columns, or if any key in the input function_kwargs is within self._forbidden_keys.

mcetl.main_gui

Provides GUIs to import data depending on the data source used, process and/or fit the data, and save everything to Excel.

@author: Donald Erb Created on May 5, 2020

Notes

The imports for the fitting and plotting guis are within their respective functions to reduce the time it takes for this module to be imported. Likewise, openpyxl is imported within _write_to_excel.

mcetl.main_gui.**SAVE_FOLDER**
The file path to the folder in which all 'previous_files_{DataSource.name}.json' files are saved. Depends on operating system.

Type pathlib.Path[348]

Module Contents

Functions

launch_main_gui (page 67)	Goes through all steps to find files, process/fit/plot the imported data, and save to Excel.

mcetl.main_gui.**launch_main_gui**(*data_sources*, *fitting_mpl_params=None*)
Goes through all steps to find files, process/fit/plot the imported data, and save to Excel.

Parameters

- **data_sources** (*list*[349] (DataSource (page 54)) *or tuple*[350] (DataSource (page 54)) *or* DataSource (page 54)) -- A list or tuple of mcetl.DataSource objects, or a single DataSource.
- **fitting_mpl_params** (*dict*[351], *optional*) -- A dictionary of changes for Matplotlib's rcParams to use during fitting. If None, will use the selected DataSource's figure_rcparams attribute.

Returns

output --

A dictionary containing the following keys and values:

[347] https://docs.python.org/3/library/exceptions.html#ValueError
[348] https://docs.python.org/3/library/pathlib.html#pathlib.Path
[349] https://docs.python.org/3/library/stdtypes.html#list
[350] https://docs.python.org/3/library/stdtypes.html#tuple
[351] https://docs.python.org/3/library/stdtypes.html#dict

'dataframes': list or None A list of lists of dataframes, with each dataframe containing the data imported from a raw data file; will be None if the function fails before importing data, or if the only processing step taken was moving files.

'fit_results': list or None A nested list of lists of lmfit.ModelResult objects, with each ModelResult pertaining to the fitting of a data entry, each list of ModelResults containing all of the fits for a single sample, and east list of lists pertaining to the data within one dataset. Will be None if fitting is not done, or only partially filled if the fitting process ends early.

'plot_results': list or None A list of lists, with one entry per dataset. Each interior list is composed of a matplotlib.Figure object and a dictionary of matplotlib.Axes objects. Will be None if plotting is not done, or only partially filled if the plotting process ends early.

'writer': pd.ExcelWriter or None The pandas ExcelWriter used to create the output Excel file; will be None if the output results were not saved to Excel.

Return type dict[352]

Notes

The entire function is wrapped in a try-except block. If the user exits the program early by exiting out of a GUI, a custom WindowCloseError exception is thrown, which is just passed, allowing the program is close without error. If other exceptions occur, their traceback is printed.

mcetl.plot_utils

Provides utility functions, classes, and constants for plotting.

Separated from utils.py to reduce import load time since matplotlib imports are not needed for base usage. Useful functions are put here in order to prevent circular importing within the other files.

@author: Donald Erb Created on Nov 11, 2020

`mcetl.plot_utils.`**`CANVAS_SIZE`**

A tuple specifying the size (in pixels) of the figure canvas used in various GUIs for mcetl. This can be modified if the user wishes a larger or smaller canvas. The default is (800, 800).

Type tuple[353](int[354], int[355])

Module Contents

Classes

EmbeddedFigure (page 69)	Class defining a PySimpleGUI window with an embedded matplotlib Figure.
PlotToolbar (page 71)	Custom toolbar without the subplots and save figure buttons.

[352] https://docs.python.org/3/library/stdtypes.html#dict
[353] https://docs.python.org/3/library/stdtypes.html#tuple
[354] https://docs.python.org/3/library/functions.html#int
[355] https://docs.python.org/3/library/functions.html#int

Functions

determine_dpi (page 71)	Gives the correct dpi to fit the figure within the GUI canvas.
draw_figure_on_canvas (page 72)	Places the figure and toolbar onto the canvas.
get_dpi_correction (page 73)	Calculates the correction factor needed to create a figure with the desired dpi.
scale_axis (page 73)	Calculates the new bounds to scale the current axis bounds.

class mcetl.plot_utils.**EmbeddedFigure**(*x*, *y*, *click_list=None*, *enable_events=True*, *enable_keybinds=True*, *toolbar_class=NavigationToolbar2Tk*)

Class defining a PySimpleGUI window with an embedded matplotlib Figure.

Class to be used to define BackgroundSelector and PeakSelector classes, which share common functions.

Parameters

- **x** (*array-like*) -- The x-values to be plotted.
- **y** (*array-like*) -- The y-values to be plotted.
- **click_list** (*list*[356], *optional*) -- A list of selected points on the plot.
- **enable_events** (*bool*[357], *optional*) -- If True (default), will connect self.events (defaults to self._on_click, self._on_pick, and self._on_key) to the figure when it is drawn on the canvas. If False, the figure will have no connected events.
- **enable_keybinds** (*bool*[358], *optional*) -- If True (default), will connect the matplotlib keybind events to the figure.
- **toolbar_class** (*NavigationToolbar2Tk, optional*) -- The class of the toolbar to place in the window. The default is NavigationToolbar2Tk.

x

The x-values to be plotted.

Type array-like

y

The y-values to be plotted.

Type array-like

click_list

A list of selected points on the plot.

Type list[359]

figure

The figure embedded in the window.

Type plt.Figure

axis

The main axis on the figure which owns the events.

[356] https://docs.python.org/3/library/stdtypes.html#list
[357] https://docs.python.org/3/library/functions.html#bool
[358] https://docs.python.org/3/library/functions.html#bool
[359] https://docs.python.org/3/library/stdtypes.html#list

Type plt.Axes

canvas
The PySimpleGUI canvas element in the window which contains the figure.

Type sg.Canvas

toolbar_canvas
The PySimpleGUI canvas element that contains the toolbar for the figure.

Type sg.Canvas

picked_object
The selected Artist objected on the figure. Useful for pick events.

Type plt.Artist

xaxis_limits
The x axis limits when the figure is first created. The values are used to determine the size of the Ellipse place by the _create_circle function. The initial limits are used so that zooming on the figure does not change the size of the Ellipse.

Type tuple[360]

yaxis_limits
The y axis limits when the figure is first created.

Type tuple[361]

events
A list containing the events for the figure. Each item in the list is a list or tuple with the first item being the matplotlib event, such as 'pick_event', and the second item is a callable (function) to be executed when the event occurs.

Type list[362](tuple[363](str[364], Callable))

canvas_size
The size, in pixels, of the figure to be created. Default is (CANVAS_SIZE[0], CANVAS_SIZE[1] - 100), which is (800, 700).

Type tuple[365](float[366], float[367])

toolbar_class
The class of the toolbar to place in the window. The default is NavigationToolbar2Tk.

Type NavigationToolbar2Tk

window
The PySimpleGUI window containing the figure.

Type sg.Window

enable_keybinds
If True, will allow using matplotlib's keybinds to trigger events on the figure.

Type bool[368]

[360] https://docs.python.org/3/library/stdtypes.html#tuple
[361] https://docs.python.org/3/library/stdtypes.html#tuple
[362] https://docs.python.org/3/library/stdtypes.html#list
[363] https://docs.python.org/3/library/stdtypes.html#tuple
[364] https://docs.python.org/3/library/stdtypes.html#str
[365] https://docs.python.org/3/library/stdtypes.html#tuple
[366] https://docs.python.org/3/library/functions.html#float
[367] https://docs.python.org/3/library/functions.html#float
[368] https://docs.python.org/3/library/functions.html#bool

Notes

This class allows easy subclassing to create simple windows with embedded matplotlib figures.

A typical __init__ for a subclass should create the figure and axes, create the window, and then place the figure within the window's canvas. For example (note that plt designates matplotlib.pyplot)

```
>>> class SimpleEmbeddedFigure(EmbeddedFigure):
        def __init__(x, y, **kwargs):
            super().__init__(x, y, **kwargs)
            self.figure, self.axis = plt.subplots()
            self.axis.plot(self.x, self.y)
            self._create_window()
            self._place_figure_on_canvas()
```

The only function that should be publically available is the event_loop method, which should return the desired output.

To close the window, use the self._close() method, which ensures that both the window and the figure are correctly closed.

Initialize self. See help(type(self)) for accurate signature.

event_loop(*self*)
Handles the event loop for the GUI.

Notes

This function should typically be overwritten by a subclass, and should typically return any desired values from the embedded figure.

This simple implementation makes the window visible, and closes the window as soon as anything in the window is clicked.

class mcetl.plot_utils.**PlotToolbar**(*fig_canvas*, *canvas*, ***kwargs*)
Bases: matplotlib.backends.backend_tkagg.NavigationToolbar2Tk

Custom toolbar without the subplots and save figure buttons.

Ensures that saving is done through the save menu in the window, which gives better options for output image quality and ensures the figure dimensions are correct. The subplots button is removed so that the user does not mess with the plot layout since it is handled by using matplotlib's tight layout.

Parameters

- **fig_canvas** (*matplotlib.FigureCanvas*) -- The figure canvas on which to operate.
- **canvas** (*tkinter.Canvas*) -- The Canvas element which owns this toolbar.
- ****kwargs*** -- Any additional keyword arguments to pass to NavigationToolbar2Tk.

mcetl.plot_utils.**determine_dpi**(*fig_height*, *fig_width*, *dpi*, *canvas_size=CANVAS_SIZE*)
Gives the correct dpi to fit the figure within the GUI canvas.

Parameters

- **fig_height** (*float*[369]) -- The figure height.
- **fig_width** (*float*[370]) -- The figure width.

[369] https://docs.python.org/3/library/functions.html#float
[370] https://docs.python.org/3/library/functions.html#float

- **dpi** (*float*[371]) -- The desired figure dpi.
- **canvas_size** (*tuple*[372]*(int*[373]*, int*[374]*), optional*) -- The size of the canvas that the figure will be placed on. Defaults to CANVAS_SIZE.

Returns The correct dpi to fit the figure onto the GUI canvas.

Return type float[375]

Notes

When not saving, the dpi needs to be scaled to fit the figure on the GUI's canvas, and that scaling is called size_scale. For example, if the desired size was 1600 x 1200 pixels with a dpi of 300, the figure would be scaled down to 800 x 600 pixels to fit onto the canvas, so the dpi would be changed to 150, with a size_scale of 0.5.

A dpi_scale correction is needed because the qt5Agg backend will change the dpi to 2x the specified dpi when the display scaling in Windows is not 100%. I am not sure how it works on non-Windows operating systems.

The final dpi when not saving is equal to dpi * size_scale * dpi_scale.

mcetl.plot_utils.**draw_figure_on_canvas**(*canvas*, *figure*, *toolbar_canvas=None*, *toolbar_class=NavigationToolbar2Tk*, *kwargs=None*)

Places the figure and toolbar onto the canvas.

Parameters

- **canvas** (*tkinter.Canvas*) -- The tkinter Canvas element for the figure.
- **figure** (*plt.Figure*) -- The figure to be place on the canvas.
- **toolbar_canvas** (*tkinter.Canvas, optional*) -- The tkinter Canvas element for the toolbar.
- **toolbar_class** (*NavigationToolbar2Tk, optional*) -- The toolbar class used to create the toolbar for the figure. The default is NavigationToolbar2Tk.
- **kwargs** (*dict*[376]*, optional*) -- Keyword arguments designating how to pack the figure into the window. Relevant keys are 'canvas' and 'toolbar', with values being dictionaries containing keyword arguments to pass to pack.

Returns

- **figure_canvas** (*FigureCanvasTkAgg or None*) -- The canvas containing the figure. Is None if drawing the canvas caused an exception.
- **toolbar** (*NavigationToolbar2Tk or None*) -- The created toolbar. Is None if toolbar_canvas is None, or if there was an error when drawing figure_canvas.

[371] https://docs.python.org/3/library/functions.html#float
[372] https://docs.python.org/3/library/stdtypes.html#tuple
[373] https://docs.python.org/3/library/functions.html#int
[374] https://docs.python.org/3/library/functions.html#int
[375] https://docs.python.org/3/library/functions.html#float
[376] https://docs.python.org/3/library/stdtypes.html#dict

Notes

The canvas children are destroyed after drawing the figure canvas so that there is a seamless transition from the old canvas to the new canvas.

The toolbar is packed before the figure canvas because the figure canvas shoulud be more flexible to resizing if they are on the same canvas.

mcetl.plot_utils.**get_dpi_correction**(*dpi*)

Calculates the correction factor needed to create a figure with the desired dpi.

Necessary because some matplotlib backends (namely qt5Agg) will adjust the dpi of the figure after creation.

Parameters **dpi** (*float*[377] *or int*[378]) -- The desired figure dpi.

Returns **dpi_correction** -- The scaling factor needed to create a figure with the desired dpi.

Return type float[379]

Notes

The matplotlib dpi correction occurs when the operating system display scaling is set to any value not equal to 100% (at least on Windows, other operating systems are unknown). This may cause issues when using UHD monitors, but I cannot test.

To get the desired dpi, simply create a figure with a dpi equal to dpi * dpi_correction.

mcetl.plot_utils.**scale_axis**(*axis_bounds*, *lower_scale=None*, *upper_scale=None*)

Calculates the new bounds to scale the current axis bounds.

The new bounds are calculated by multiplying the desired scale factor by the current difference of the upper and lower bounds, and then adding (for upper bound) or subtracting (for lower bound) from the current bound.

Parameters

- **axis_bounds** (*tuple*[380] *(float*[381]*, float*[382]*)*) -- The current lower and upper axis bounds.
- **lower_scale** (*float*[383]*, optional*) -- The desired fraction by which to scale the current lower bound. If None, will not scale the current lower bounds.
- **upper_scale** (*float*[384]*, optional*) -- The desired fraction by which to scale the current upper bound. If None, will not scale the current upper bounds.

Returns

- **lower_bound** (*float*) -- The lower bound after scaling.
- **upper_bound** (*float*) -- The upper bound after scaling.

[377] https://docs.python.org/3/library/functions.html#float
[378] https://docs.python.org/3/library/functions.html#int
[379] https://docs.python.org/3/library/functions.html#float
[380] https://docs.python.org/3/library/stdtypes.html#tuple
[381] https://docs.python.org/3/library/functions.html#float
[382] https://docs.python.org/3/library/functions.html#float
[383] https://docs.python.org/3/library/functions.html#float
[384] https://docs.python.org/3/library/functions.html#float

mcetl.raw_data

Creates folders and files with simulated data for various characterization techniques

Notes

All data is made up and does not correspond to the materials listed. The data is meant to simply emulate real data and allow for basic analysis.

@author: Donald Erb Created on Jun 15, 2020

Module Contents

Functions

generate_raw_data (page 74)	Generates data for all of the techniques in this file.

mcetl.raw_data.**generate_raw_data**(*directory=None*, *num_files=None*, *show_plots=False*)

Generates data for all of the techniques in this file.

Convenience function to generate data for all techniques rather that calling the functions one at a time.

Parameters

- **directory** (*str*[385], *optional*) -- The file path to place the Raw Data folder.
- **num_files** (*int*[386], *optional*) -- The number of files to create per characterization technique.
- **show_plots** (*bool*[387], *optional*) -- If True, will show plots of the created data. If False (default), will close the created figures and not show the plots.

Notes

Currently supported characterization techniques include: XRD, FTIR, Raman, TGA, DSC, Rheometry, Uniaxial tensile test, Pore Size Analysis

mcetl.utils

Provides utility functions, classes, and constants.

Useful functions are put here in order to prevent circular importing within the other files.

The functions contained within this module ease the use of user-interfaces, selecting options for opening files, and working with Excel.

@author: Donald Erb Created on Jul 15, 2020

mcetl.utils.**PROCEED_COLOR**

The button color for all buttons that proceed to the next window. The default is ('white', '#00A949'), where '#00A949' is a bright green.

[385] https://docs.python.org/3/library/stdtypes.html#str
[386] https://docs.python.org/3/library/functions.html#int
[387] https://docs.python.org/3/library/functions.html#bool

Type tuple[388](str[389], str[390])

Module Contents

Functions

check_availability (page 75)	Checks whether an optional dependency is available to import.
doc_lru_cache (page 76)	Decorator that allows keeping a function's docstring when using functools.lru_cache.
excel_column_name (page 76)	Converts 1-based index to the Excel column name.
get_min_size (page 77)	Returns the minimum size for a GUI element to match the screen size.
open_multiple_files (page 77)	Creates a prompt to open multiple files and add their contents to a dataframe.
optimize_memory (page 77)	Optimizes dataframe memory usage by converting data types.
raw_data_import (page 78)	Used to import data from the specified file into pandas DataFrames.
safely_close_window (page 78)	Closes a PySimpleGUI window and removes the window and its layout.
select_file_gui (page 78)	GUI to select a file and input the necessary options to import its data.
series_to_numpy (page 79)	Tries to convert a pandas Series to a numpy array with the desired dtype.
set_dpi_awareness (page 80)	Sets DPI awareness for Windows operating system so that GUIs are not blurry.
show_dataframes (page 80)	Used to show data to help select the right columns or datasets from the data.
string_to_unicode (page 80)	Converts strings to unicode by replacing `'\\'` with `'\'`.
stringify_backslash (page 81)	Fixes strings containing backslash, such as `'\n'`, so that they display properly in GUIs.
validate_inputs (page 81)	Validates entries from a PySimpleGUI window and converts to the desired type.
validate_sheet_name (page 82)	Ensures that the desired Excel sheet name is valid.

exception mcetl.utils.**WindowCloseError**

Bases: Exception[391]

Custom exception to allow exiting a GUI window to stop the program.

Initialize self. See help(type(self)) for accurate signature.

with_traceback()

Exception.with_traceback(tb) -- set self.__traceback__ to tb and return self.

mcetl.utils.**check_availability**(*module*)

Checks whether an optional dependency is available to import.

[388] https://docs.python.org/3/library/stdtypes.html#tuple
[389] https://docs.python.org/3/library/stdtypes.html#str
[390] https://docs.python.org/3/library/stdtypes.html#str
[391] https://docs.python.org/3/library/exceptions.html#Exception

Does not check the module version since it is assumed that the parent module will do a version check if the module is actually usable.

Parameters **module** (*str*[392]) -- The name of the module.

Returns True if the module can be imported, False if it cannot.

Return type bool[393]

Notes

It is faster to use importlib to check the availability of the module rather than doing a try-except block to try and import the module, since importlib does not actually import the module.

mcetl.utils.**doc_lru_cache**(*function=None*, ***lru_cache_kwargs*)

Decorator that allows keeping a function's docstring when using functools.lru_cache.

Parameters

- **function** (*Callable*) -- The function to use. If used as a decorator and lru_cache_kwargs are specified, then function will be None.
- ****lru_cache_kwargs** -- Any keyword arguments to pass to functools.lru_cache (maxsize and/or typed, as of Python 3.9).

Examples

A basic usage of this decorator would look like:

```
>>> @doc_lru_cache(maxsize=200)
    def function(arg, kwarg=1)
        return arg + kwarg
```

mcetl.utils.**excel_column_name**(*index*)

Converts 1-based index to the Excel column name.

Parameters **index** (*int*[394]) -- The column number. Must be 1-based, ie. the first column number is 1 rather than 0.

Returns **col_name** -- The column name for the input index, eg. an index of 1 returns 'A'.

Return type str[395]

Raises **ValueError**[396] -- Raised if the input index is not in the range 1 <= index <= 18278, meaning the column name is not within 'A'...'ZZZ'.

[392] https://docs.python.org/3/library/stdtypes.html#str
[393] https://docs.python.org/3/library/functions.html#bool
[394] https://docs.python.org/3/library/functions.html#int
[395] https://docs.python.org/3/library/stdtypes.html#str
[396] https://docs.python.org/3/library/exceptions.html#ValueError

Notes

Caches the result so that any repeated index lookups are faster, and uses recursion to make better usage of the cache.

chr(64 + remainder) converts the remainder to a character, where 64 denotes ord('A') - 1, so if remainder = 1, chr(65) = 'A'.

mcetl.utils.**get_min_size**(*default_size*, *scale*, *dimension='both'*)
Returns the minimum size for a GUI element to match the screen size.

Parameters

- **default_size** (*int*[397]) -- The default number of pixels to use. Needed because sg.Window.get_screen_size() can return the total screen size when using multiple screens on some linux systems.
- **scale** (*float*[398]) -- The scale factor to apply to the screen size as reported by sg.Window.get_screen_size. For example, if the element size was desired to be at most 50% of the minimum screen dimension, then the scale factor is 0.5.
- **dimension** (*str*[399]) -- The screen dimension to compare. Can be either 'width', 'height', or 'both'.

Returns The minimum pixel count among scale * screen height, scale * screen width, and default_size.

Return type int[400]

mcetl.utils.**open_multiple_files**()
Creates a prompt to open multiple files and add their contents to a dataframe.

Returns **dataframes** -- A list of dataframes containing the imported data from the selected files.

Return type list[401]

mcetl.utils.**optimize_memory**(*dataframe*, *convert_objects=False*)
Optimizes dataframe memory usage by converting data types.

Optimizes object dtypes by trying to convert to other dtypes, if the pandas version is greater than 1.0.0. Optimizes numerical dtypes by downcasting to the most appropriate dtype.

Parameters

- **dataframe** (*pd.DataFrame*) -- The dataframe to optimize.
- **convert_objects** (*bool*[402]*, optional*) -- If True, will attempt to convert columns with object dtype if the pandas version is >= 1.0.0.

Returns **dataframe** -- The memory-optimized dataframe.

Return type pd.DataFrame

[397] https://docs.python.org/3/library/functions.html#int
[398] https://docs.python.org/3/library/functions.html#float
[399] https://docs.python.org/3/library/stdtypes.html#str
[400] https://docs.python.org/3/library/functions.html#int
[401] https://docs.python.org/3/library/stdtypes.html#list
[402] https://docs.python.org/3/library/functions.html#bool

Notes

convert_objects is needed because currently, when object columns are converted to a dtype of string, the row becomes a StringArray object, which does not have the tolist() method curently implemented (as of pandas version 1.0.5). openpyxl's dataframe_to_rows method uses each series's series.values.tolist() method to convert the dataframe into a generator of rows, so having a StringArray row without a tolist method causes an exception when using openpyxl's dataframe_to_rows.

Do not convert object dtypes to pandas's Int and Float dtypes since they do not mesh well with other modules.

Iterate through columns one at a time rather using dataframe.select_dtypes so that each column is overwritten immediately, rather than making a copy of all the selected columns, reducing memory usage.

mcetl.utils.**raw_data_import**(*window_values*, *file*, *show_popup=True*)

Used to import data from the specified file into pandas DataFrames.

Also used to show how data will look after using certain import values.

Parameters

- **window_values** (*dict*[403]) -- A dictionary with keys 'row_start', 'row_end', columns', 'separator', and optionally 'sheet'.
- **file** (*str*[404] *or pathlib.Path*[405] *or pd.ExcelFile*) -- A string or Path for the file to be imported, or a pandas ExcelFile, to use for reading spreadsheet data.
- **show_popup** (*bool*[406]) -- If True, will display a popup window showing a table of the data.

Returns **dataframes** -- A list of dataframes containing the data after importing if show_popup is False, otherwise returns None.

Return type list[407](pd.DataFrame) or None[408]

Notes

If using a spreadsheet format ('xls', 'xlsx', 'odf', etc.), allows using any of the available engines for pandas.read_excel, and will just let pandas notify the user if the proper engine is not installed.

Optimizes the memory usage of the imported data before returning.

mcetl.utils.**safely_close_window**(*window*)

Closes a PySimpleGUI window and removes the window and its layout.

Used when exiting a window early by manually closing the window. Ensures that the window is properly closed and then raises a WindowCloseError exception, which can be used to determine that the window was manually closed.

Parameters **window** (*sg.Window*) -- The window that will be closed.

Raises ***WindowCloseError*** (page 75) -- Custom exception to notify that the window has been closed earlier than expected.

mcetl.utils.**select_file_gui**(*data_source=None*, *file=None*, *previous_inputs=None*, *assign_columns=False*)

GUI to select a file and input the necessary options to import its data.

[403] https://docs.python.org/3/library/stdtypes.html#dict
[404] https://docs.python.org/3/library/stdtypes.html#str
[405] https://docs.python.org/3/library/pathlib.html#pathlib.Path
[406] https://docs.python.org/3/library/functions.html#bool
[407] https://docs.python.org/3/library/stdtypes.html#list
[408] https://docs.python.org/3/library/constants.html#None

Parameters

- **data_source** (DataSource (page 54), *optional*) -- The DataSource object used for opening the file.
- **file** (*str*[409], *optional*) -- A string containing the path to the file to be imported.
- **previous_inputs** (*dict*[410], *optional*) -- A dictionary containing the values from a previous usage of this function, that will be used to overwrite the defaults. Note, if opening Excel files, the previous_inputs will have no effect.
- **assign_columns** (*bool*[411], *optional*) -- If True, designates that the columns for each unique variable in the data source need to be identified. If False (or if data_source is None), then will not prompt user to select columns for variables.

Returns **values** -- A dictionary containing the items necessary for importing data from the selected file.

Return type dict[412]

Notes

If using a spreadsheet format ('xls', 'xlsx', 'odf', etc.), allows using any of the available engines for pandas.read_excel, and will just let pandas notify the user if the proper engine is not installed. The file selection window, however, will only show 'xlsx', 'xlsm', 'csv', 'txt', and potentially 'xls', so that users are not steered towards selecting a format that does not work with the default mcetl libraries.

mcetl.utils.**series_to_numpy**(*series*, *dtype=float*)

Tries to convert a pandas Series to a numpy array with the desired dtype.

If initial conversion does not work, tries to convert series to object first. If that is not successful and if the first item is a string, assumes the first item is a header, converts it to None, and tries the conversion. If that is still unsuccessful, then an array of the series is returned without changing the dtype.

Parameters

- **series** (*pd.Series*) -- The series to convert to numpy with the desired dtype.
- **dtype** (*type*[413], *optional*) -- The dtype to use in the numpy array of the series. Default is float.

Returns **output** -- The input series with the specified dtype if conversion was successful. Otherwise, the output is an ndarray of the input series without dtype conversion.

Return type np.ndarray

[409] https://docs.python.org/3/library/stdtypes.html#str
[410] https://docs.python.org/3/library/stdtypes.html#dict
[411] https://docs.python.org/3/library/functions.html#bool
[412] https://docs.python.org/3/library/stdtypes.html#dict
[413] https://docs.python.org/3/library/functions.html#type

Notes

This function is needed because pandas's pd.NA and extension arrays do not work well with other modules and can be difficult to convert.

mcetl.utils.**set_dpi_awareness**(*awareness_level=1*)
Sets DPI awareness for Windows operating system so that GUIs are not blurry.

Fixes blurry tkinter GUIs due to weird dpi scaling in Windows os. Other operating systems are ignored.

Parameters **awareness_level** (*{1, 0, 2}*) -- The dpi awareness level to set. 0 turns off dpi awareness, 1 sets dpi awareness to scale with the system dpi and automatically changes when the system dpi changes, and 2 sets dpi awareness per monitor and does not change when system dpi changes. Default is 1.

Raises **ValueError**[414] -- Raised if awareness_level is not 0, 1, or 2.

Notes

Will only work on Windows 8.1 or Windows 10. Not sure if earlier versions of Windows have this issue anyway.

mcetl.utils.**show_dataframes**(*dataframes*, *title='Raw Data'*)
Used to show data to help select the right columns or datasets from the data.

Parameters

- **dataframes** (*list*[415] *or pd.DataFrame*) -- Either (1) a pandas DataFrame, (2) a list of DataFrames, or (3) a list of lists of DataFrames. The layout of the window will depend on the input type.
- **title** (*str*[416]*, optional*) -- The title for the popup window.

Returns **window** -- If no exceptions occur, a PySimpleGUI window will be returned; otherwise, None will be returned.

Return type sg.Window or None[417]

mcetl.utils.**string_to_unicode**(*input_list*)
Converts strings to unicode by replacing `'\\'` with `'\'`.

Necessary because user input from text elements in GUIs are raw strings and will convert any `'\'` input by the user to `'\\'`, which will not be converted to the desired unicode. If the string already has unicode characters, it will be left alone.

Also converts things like `'\\n'` and `'\\t'` to `'\n'` and `'\t'`, respectively, so that inputs are correctly interpreted.

Parameters **input_list** (*(list*[418]*, tuple*[419]*) or str*[420]) -- A container of strings or a single string.

Returns **output** -- A container of strings or a single string, depending on the input, with the unicode correctly converted.

[414] https://docs.python.org/3/library/exceptions.html#ValueError
[415] https://docs.python.org/3/library/stdtypes.html#list
[416] https://docs.python.org/3/library/stdtypes.html#str
[417] https://docs.python.org/3/library/constants.html#None
[418] https://docs.python.org/3/library/stdtypes.html#list
[419] https://docs.python.org/3/library/stdtypes.html#tuple
[420] https://docs.python.org/3/library/stdtypes.html#str

Return type (list[421], tuple[422]) or str[423]

Notes

Uses raw_unicode_escape encoding to ensure that any existing unicode is correctly decoded; otherwise, it would translate incorrectly.

If using mathtext in matplotlib and want to do something like `$\nu$`, input `$\\nu$` in the GUI, which gets converted to `$\\\\nu$` by the GUI, and in turn will be converted back to `$\\nu$` by this fuction, which matplotlib considers equivalent to `$\nu$`.

mcetl.utils.**stringify_backslash**(*input_string*)

Fixes strings containing backslash, such as `'\n'`, so that they display properly in GUIs.

Parameters **input_string** (*str*[424]) -- The string that potentially contains a backslash character.

Returns **output_string** -- The string after replacing various backslash characters with their double backslash versions.

Return type str[425]

Notes

It is necessary to replace multiple characters because things like `'\n'` are considered unique characters, so simply replacing the `'\'` would not work.

mcetl.utils.**validate_inputs**(*window_values*, *integers=None*, *floats=None*, *strings=None*, *user_inputs=None*, *constraints=None*)

Validates entries from a PySimpleGUI window and converts to the desired type.

Parameters

- **window_values** (*dict*[426]) -- A dictionary of values from a PySimpleGUI window, generated by using window.read().
- **integers** (*list*[427], *optional*) -- A list of lists (see Notes below), with each key corresponding to a key in the window_values dictionary and whose values should be integers.
- **floats** (*list*[428], *optional*) -- A list of lists (see Notes below), with each key corresponding to a key in the window_values dictionary and whose values should be floats.
- **strings** (*list*[429], *optional*) -- A list of lists (see Notes below), with each key corresponding to a key in the window_values dictionary and whose values should be non-empty strings.
- **user_inputs** (*list*[430], *optional*) -- A list of lists (see Notes below), with each key corresponding to a key in the window_values dictionary and whose values should be a certain data type; the values are first determined by separating each value using ',' (default) or the last index.

[421] https://docs.python.org/3/library/stdtypes.html#list
[422] https://docs.python.org/3/library/stdtypes.html#tuple
[423] https://docs.python.org/3/library/stdtypes.html#str
[424] https://docs.python.org/3/library/stdtypes.html#str
[425] https://docs.python.org/3/library/stdtypes.html#str
[426] https://docs.python.org/3/library/stdtypes.html#dict
[427] https://docs.python.org/3/library/stdtypes.html#list
[428] https://docs.python.org/3/library/stdtypes.html#list
[429] https://docs.python.org/3/library/stdtypes.html#list
[430] https://docs.python.org/3/library/stdtypes.html#list

- **constraints** (*list*[431], *optional*) -- A list of lists (see Notes below), with each key corresponding to a key in the window_values dictionary and whose values should be ints or floats constrained between upper and lower bounds.

Returns True if all data in the window_values dictionary is correct. False if there is any error with the values in the window_values dictionary.

Return type bool[432]

Notes

Inputs for integers, floats, and strings are [[key, display text],].

For example: [['peak_width', 'peak width']]

Inputs for user_inputs are [[key, display text, data type, allow_empty_input (optional), separator (optional)],],

where separator is a string, and allow_empty_input is a boolean. If no separator is given, it is assumed to be a comma (','), and if no allow_empty_input value is given, it is assumed to be False. user_inputs can also be used to run the inputs through a function by setting the data type to a custom function. Use None as the separator if only a single value is wanted. For example: [

['peak_width', 'peak width', float], # ensures each entry is a float ['peak_width_2', 'peak width 2', int, False, ';'], # uses ';' as the separator ['peak_width_3', 'peak width 3', function, False, None], # no separator, verify with function ['peak_width_4', 'peak width 4', function, True, None] # allows empty input

]

Inputs for constraints are [[key, display text, lower bound, upper bound (optional)],],

where lower and upper bounds are strings with the operator and bound, such as "> 10". If lower bound or upper bound is None, then the operator and bound is assumed to be >=, -np.inf and <=, np.inf, respectively. For example: [

['peak_width', 'peak width', '> 10', '< 20'], # 10 < peak_width < 20 ['peak_width_2', 'peak width 2', None, '<= 5'] # -inf <= peak_width_2 <= 5 ['peak_width_3', 'peak width 3', '> 1'] # 1 < peak_width_2 <= inf

]

The display text will be the text that is shown to the user if the value of window_values[key] fails the validation.

#TODO eventually collect all errors so they can all be fixed at once.

mcetl.utils.**validate_sheet_name**(*sheet_name*)

Ensures that the desired Excel sheet name is valid.

Parameters **sheet_name** (*str*[433]) -- The desired sheet name.

Returns **sheet_name** -- The input sheet name. Only returned if it is valid.

Return type str[434]

Raises **ValueError**[435] -- Raised if the sheet name is greater than 31 characters or if it contains any of the following: `\, /, ?, *, [, ], :`

[431] https://docs.python.org/3/library/stdtypes.html#list
[432] https://docs.python.org/3/library/functions.html#bool
[433] https://docs.python.org/3/library/stdtypes.html#str
[434] https://docs.python.org/3/library/stdtypes.html#str
[435] https://docs.python.org/3/library/exceptions.html#ValueError

API reference documentation was auto-generated using sphinx-autoapi[436].

[436] https://github.com/readthedocs/sphinx-autoapi

CHAPTER

SIX

GALLERY

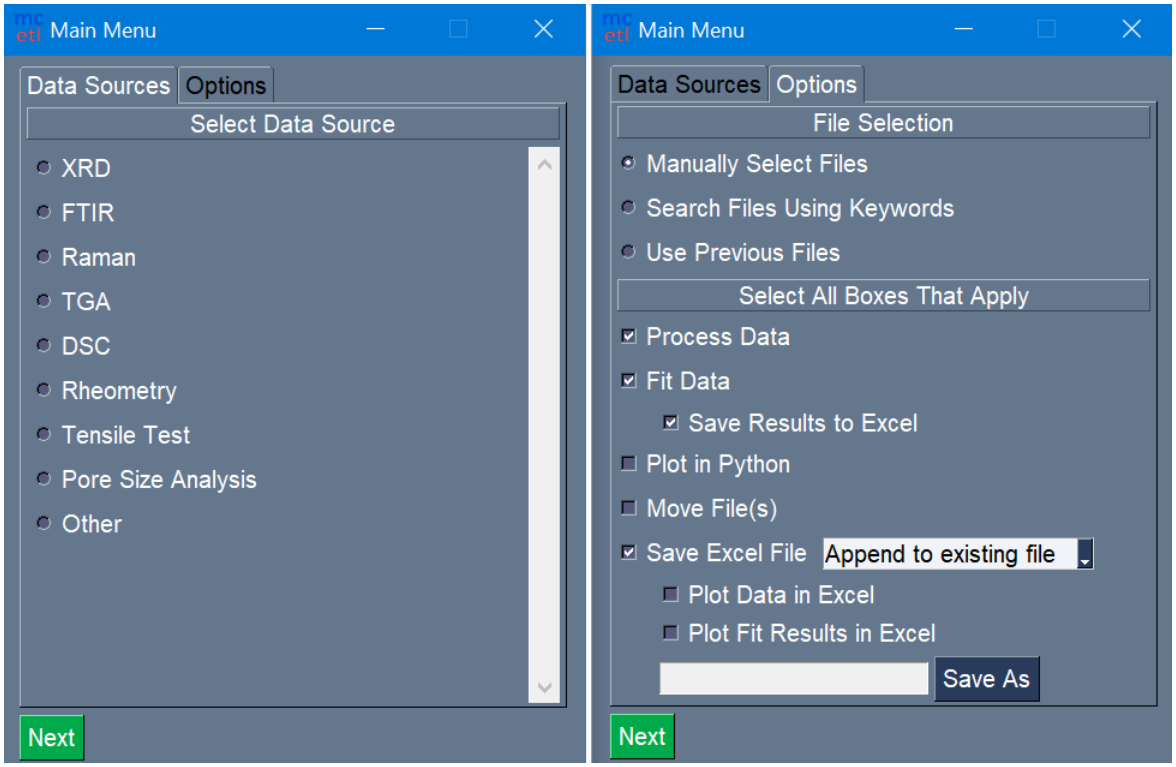

Fig. 1: Selection of DataSource and processing steps for main GUI.

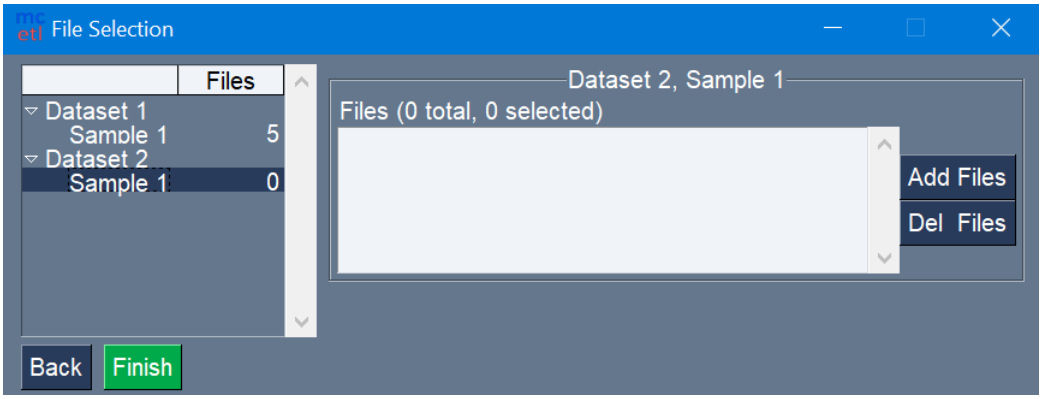

Fig. 2: File selection for each sample in each dataset.

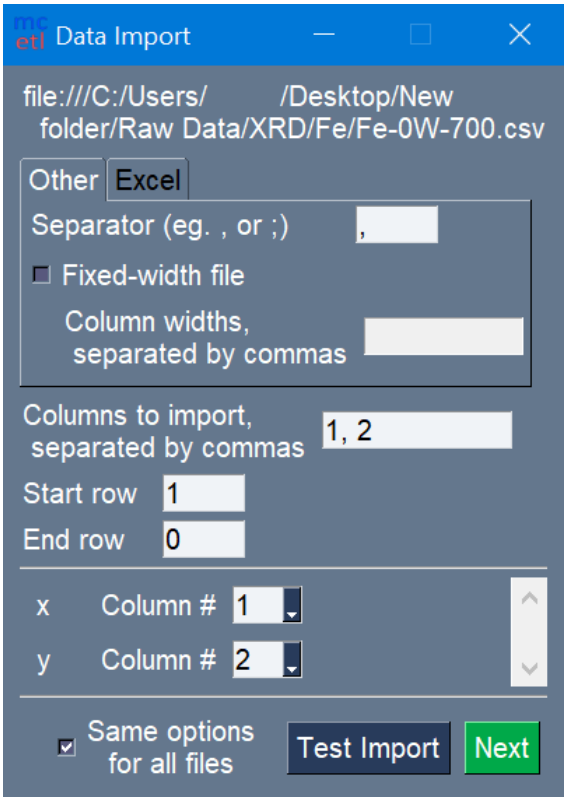

Fig. 3: Options for importing raw data.

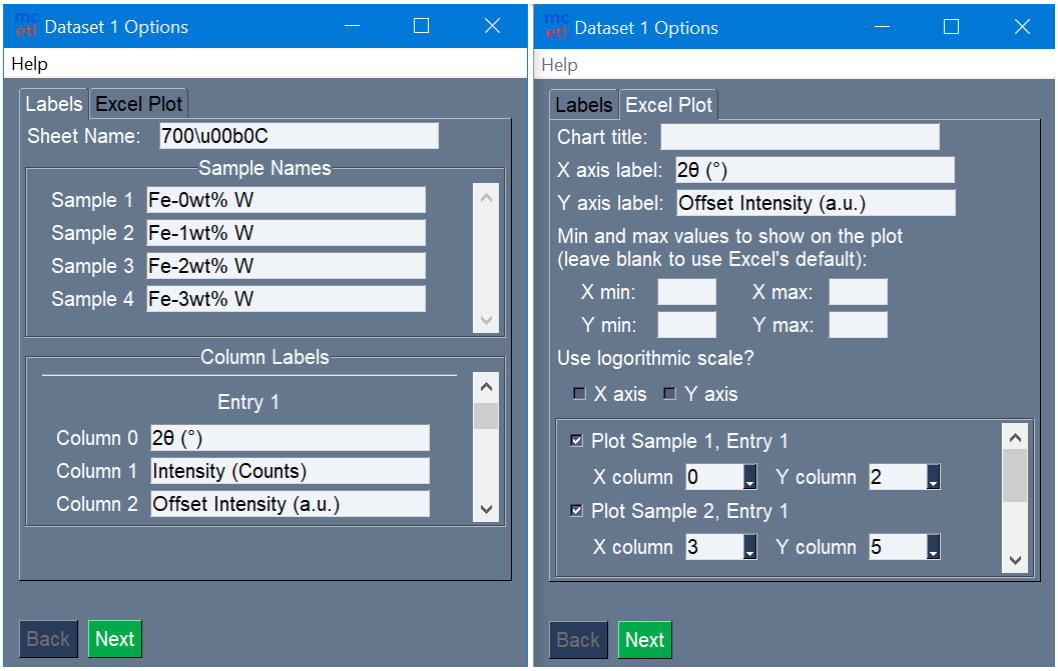

Fig. 4: Naming of samples and columns, and setup for Excel plot.

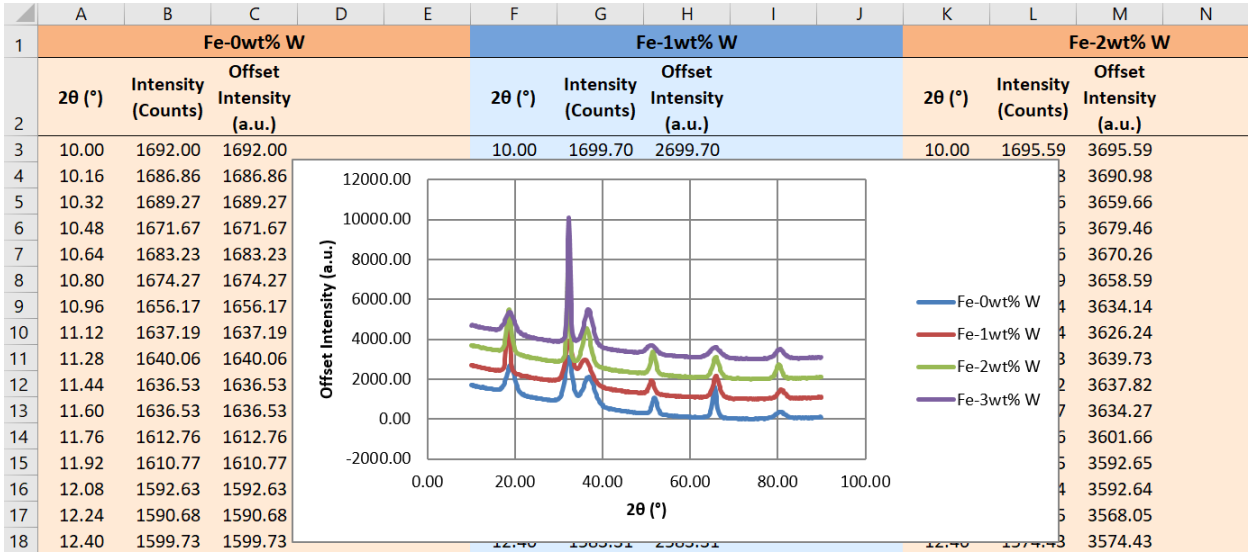

Fig. 5: The output Excel file after processing all the raw data files.

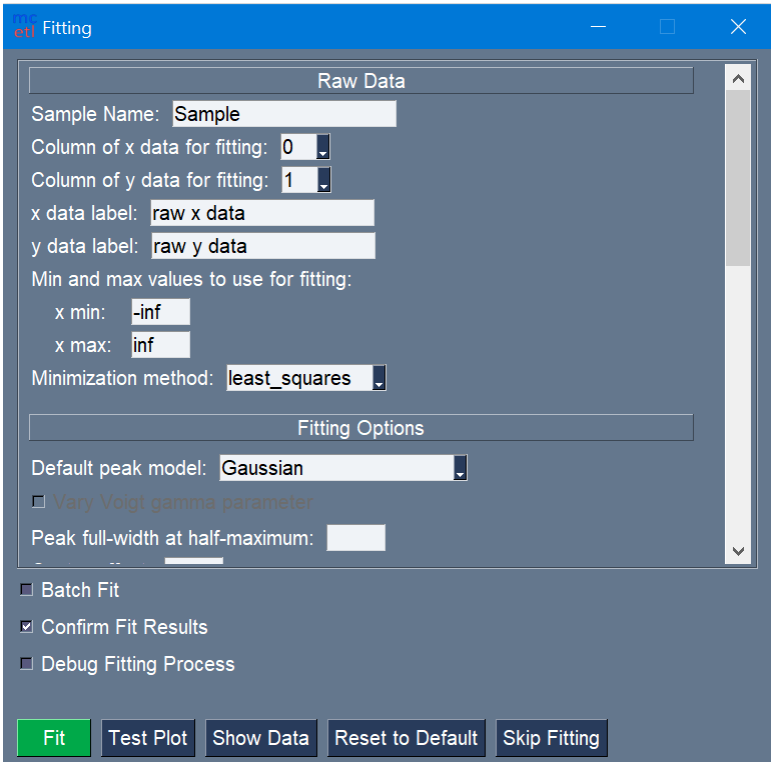

Fig. 6: The fitting GUI.

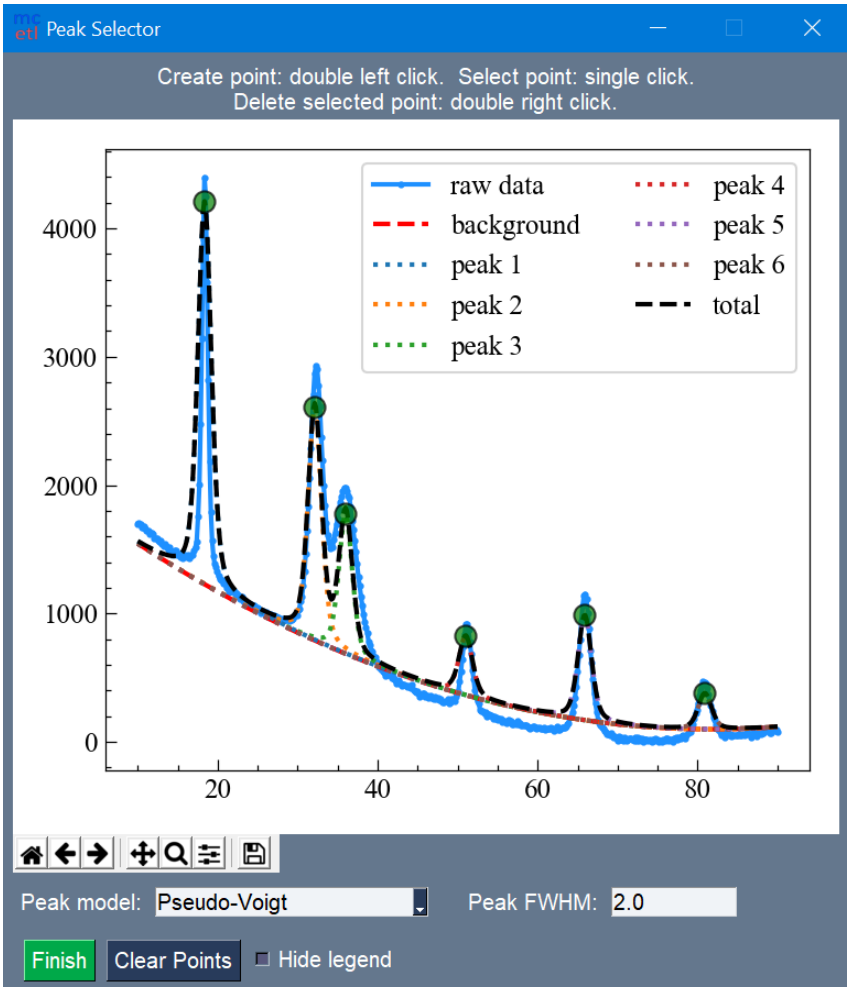

Fig. 7: Manual selection of peaks within the data.

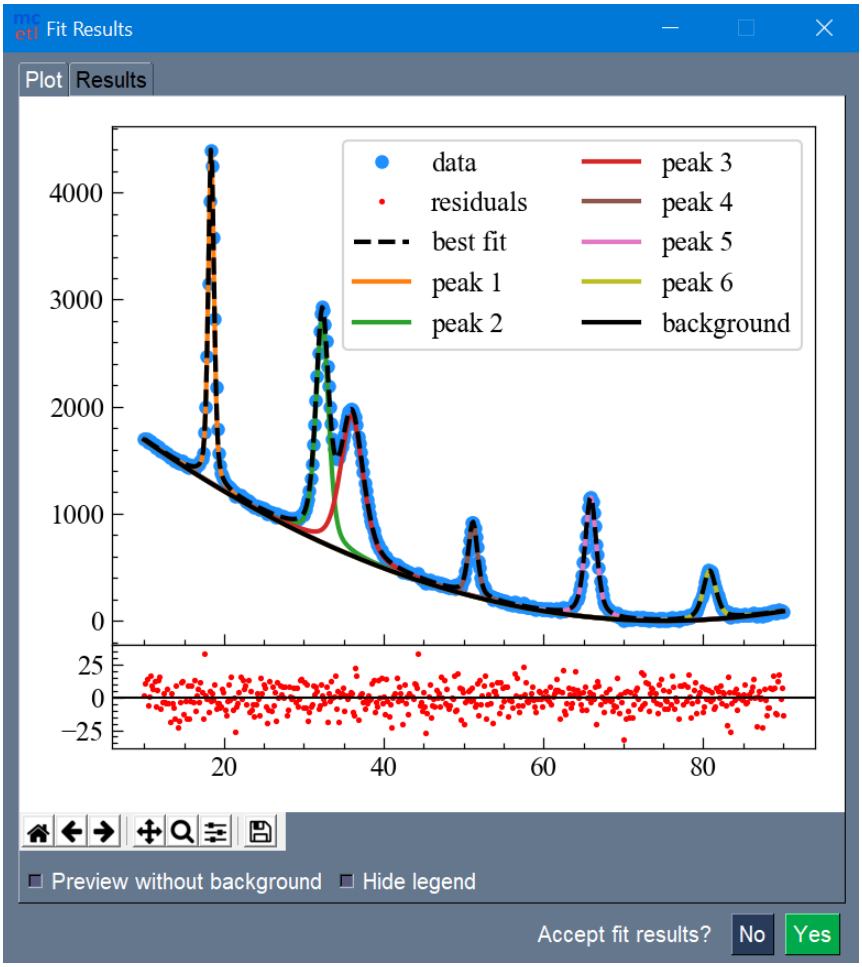

Fig. 8: Fit results with best fit and individual peaks.

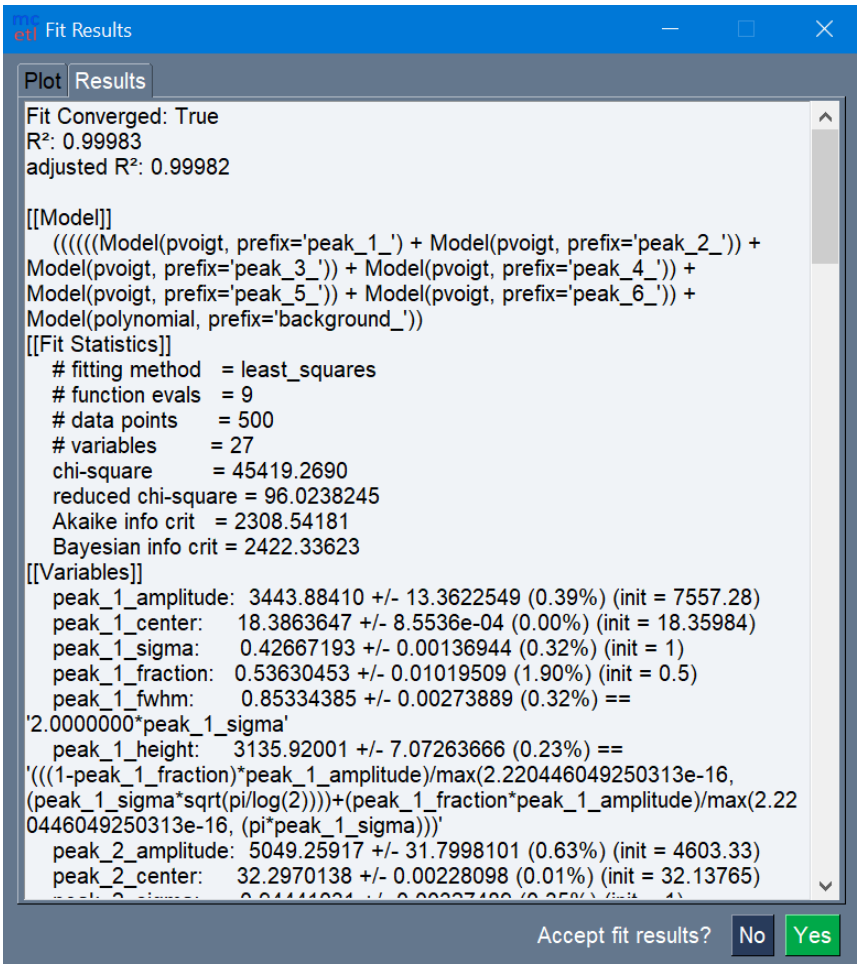

Fig. 9: Fit report for the fitting.

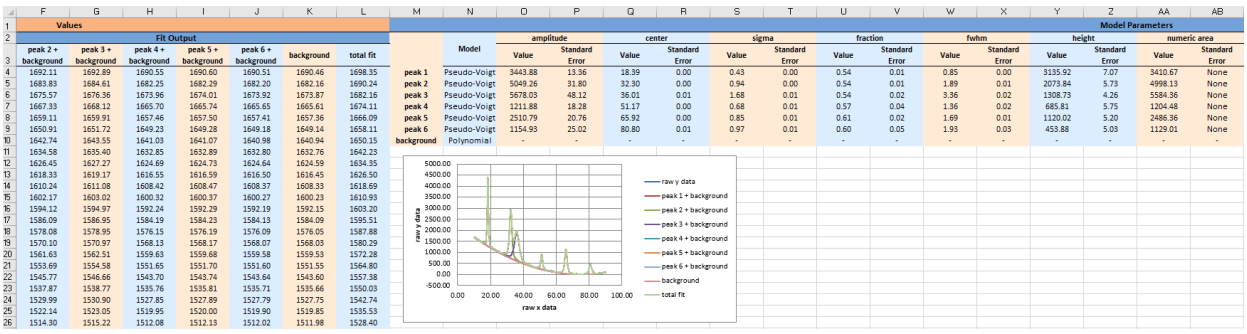

Fig. 10: The output Excel file after fitting.

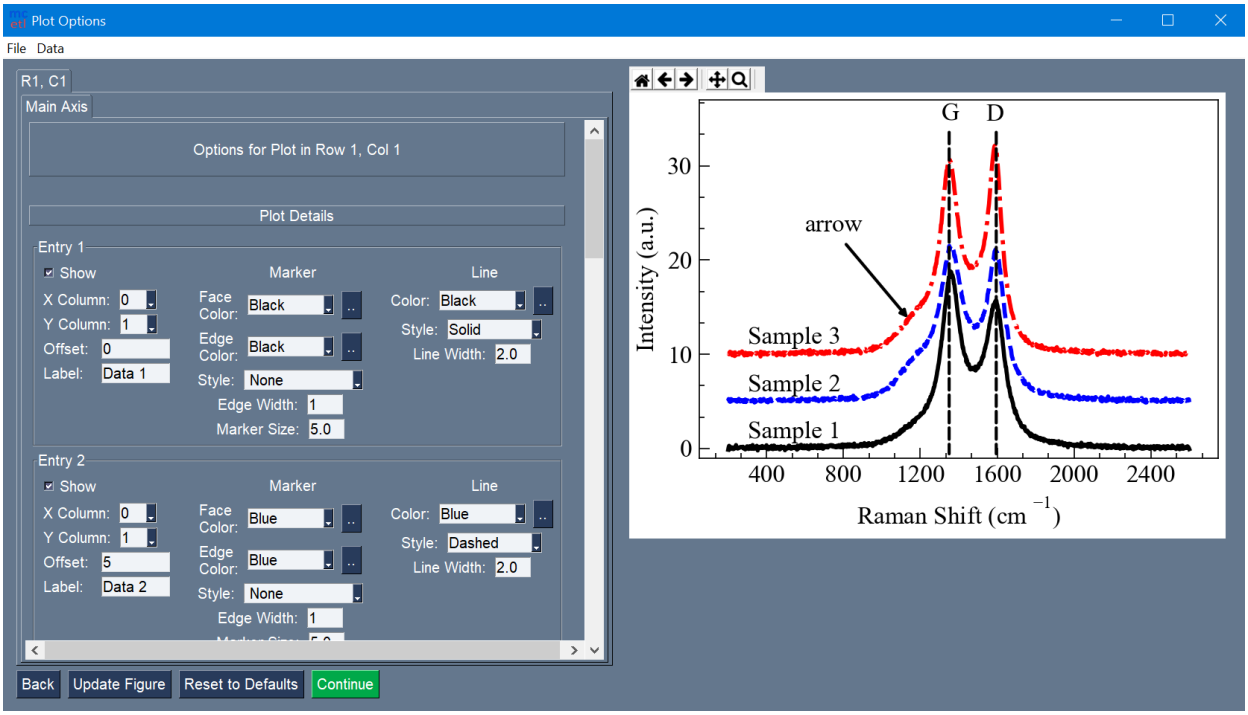

Fig. 11: The plotting GUI.

CHAPTER

SEVEN

CONTRIBUTING

Contributions are welcomed and greatly appreciated.

7.1 Bugs Reports/Feedback

Report bugs or give feedback by filing an issue at https://github.com/derb12/mcetl/issues.

If you are reporting a bug, please include:

- Your operating system, Python version, and mcetl version.
- Any further details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce/identify the bug, including the code used and any tracebacks.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible to make it easier to implement.

7.2 Pull Requests

Pull requests are welcomed for this project, but please note that unsolicited pull requests are discouraged. Please file an issue first, so that details can be discussed/finalized before a pull request is created.

Any new code or documentation must be unique (not copied from somewhere else) and able to be covered by the BSD 3-clause license. Further, any new code should follow PEP 8[437] standards as closely as possible and be fully documented using Numpy style[438] docstrings. If implementing a new feature, please provide documentation discussing its implementation, and any necessary tests.

To clone the GitHub repository and install the needed libraries for development:

```
git clone https://github.com/derb12/mcetl.git
pip -r install mcetl/requirements/requirements-development.txt
```

When submitting a pull request, follow similar procedures for a feature request, namely:

- Explain in detail how it works.
- Keep the scope as narrow as possible to make it easier to incorporate.

[437] https://www.python.org/dev/peps/pep-0008
[438] https://numpydoc.readthedocs.io/en/latest/format.html#docstring-standard

CHAPTER

EIGHT

CHANGELOG

8.1 Version 0.4.1 (2021-01-26)

This is a minor patch with bug fixes.

8.1.1 Bug Fixes

- Fixed grouping of column labels for the main GUI when there are more than 9 entries for a sample.
- Fixed processing of regex separators, such as `\s+` when importing data.

8.2 Version 0.4.0 (2021-01-24)

This is a minor version with new features, bug fixes, deprecations, and documentation improvements.

8.2.1 New Features

- The data import window extended the support of importing from spreadsheet-like files beyond just xlsx and now supports importing from any spreadsheet-like file format supported by pandas.read_excel (eg. xls, xlsm, xlsb, ods, etc). For spreadsheet formats not supported by openpyxl, and thus not supported by the default mcetl installation, it will require installing the appropriate library. For example, to read xls files, the xlrd library would need to be installed.
- Greatly improved load time of spreadsheets when using the data import window. Now, only one sheet is loaded into memory at a time (if supported by the engine used for reading the spreadsheet).
- The data import window now supports importing from fixed-width files.
- Column labels and Excel plot indices are now specified individually when using the main GUI. This allows processing uneven-sized data and working with already processed data.
- Column labels are now added to the dataframes in launch_main_gui if any processing step is specified besides moving files.
- Excel styles no longer have to be NamedStyles, allowing not adding styles to the output Excel workbook. Also, non-named styles are ~50% faster for writing to Excel.
- Added test_excel_styles method to DataSource, to allow testing whether the input styles will create valid Excel styles.
- Added a more usable window for manual file selection, which is used for launch_main_gui launch_fitting_gui, and launch_plotting_gui.

- The main GUI will now save the previous search for each DataSource separately, with a filename of previous_files_(DataSource.name).json (eg. prevous_files_XRD.json).
- Allow using all models available from lmfit and mcetl to use as the background for fitting. The few models that need kwargs during initialization now allow selecting them in the GUI.
- Added several functions to mcetl.fitting.fitting_utils to allow using the lmfit model names within code while displaying shortened versions in GUIs. Also, a full list of models available through lmfit and mcetl are generated at runtime, so that the available models match for different versions.
- Created an ExcelWriterHandler class, which is used to correctly handle opening and saving existing Excel files and to apply styles to the Excel workbook.
- Allow specifying changes to Matplotlib's rcParams for fitting when using launch_main_gui. If None are specified, will use the selected DataSource's figure_rcparams attribute.
- Allow columns to be removed by PreprocessFunctions, which also correctly updates the unique variable references.
- When importing data, columns of the imported data are now sorted following the user-input column numbers (eg. if the user-specified columns were 1, 0, 2, then the data would first be imported with columns ordered as 0, 1, 2 and then the columns are rearranged to 1, 0, 2. Note that column names are still specified as ascending numbers, so even though the data corresponds to columns 1, 0, 2 in the data file, the column names in the pandas DataFrame are still 0, 1, 2). Note that this feature was already implemented when importing from spreadsheets, but is now extended to all other formats.
- Allow using Matplotlib keybindings for most embedded figures, such as within the fitting GUI. This allows doing things like switching axis scales from linear to log.
- Added more fit descriptors to the saved Excel file from fitting, such as whether the fit converged and the number of independant variables.
- Added numerical calculations for area, extremum, mode, and full-width-at-half-maximum when doing data fitting. These calculations are performed on the peak models after fitting, and are included to at least give an estimate of their values for models that do not have analytical solutions, like Breit-Wigner-Fano, Moffat, and skewed Gaussian/Voigt.
- Made the file import options more flexible by allowing user to select whether or not to use same import options for all files (option is ignored for spreadsheet-like files). Also allow inputting previous inputs for the file import function, so that any changes to the defaults can be maintained, especially helpful when manually opening files for fitting or plotting.
- Added a modified Breit-Wigner-Fano peak model for fitting.
- The fitting GUI now allows confirmation of the fit results before proceeding. A plot of the fit results and a printed out fit report are given, and the user can select to go back and redo the fit.
- Allow inputting Excel styles in launch_fitting_gui so that custom styles can be used when writing fit results to Excel when using launch_fitting_gui directly.
- Created a EmbeddedFigure class in plot_utils for easily creating windows with embedded Matplotlib figures that optionally have events. Is easily subclassed to create custom windows with embedded figures and event loops.

8.2.2 Bug Fixes

- DPI awareness is no longer set immediately upon importing mcetl on Windows operating systems. That way, users can decide whether or not to set dpi awareness. To set dpi awareness, users must call mcetl.set_dpi_awareness() before opening any GUIs.
- Correctly handle PermissionError for the main GUI when deciding which folder to save file searches to and when writing file searches to file. PermissionError is still raised if read access is not granted, so that user is made aware that they need to set the folder to something that grants access.
- Added a function in utils for converting pandas series to numpy with a specific dtype to reduce the chance of error occuring during conversion.
- Fixed an issue when plotting in Excel with a log scale if one of the specified bounds was <= 0.
- The data import window will only attempt to assign indices for a DataSource's unique variables if processing.
- The entry and sample separation columns that are added when using the main GUI if processing and writing to Excel are now removed when splitting the merged dataframes back into individual entries, so that the returned DataFrames contain only the actual data.
- Ensured that Excel sheet names are valid.
- Simplified writing to csv for plotting GUI. Removed the column indices when reading/writing csv data within the plotting GUI. Now, columns are just directly taken from the data.
- Switched to using df.iloc[:, col_number] rather than df[col_number] to get columns by their indices so that dataframes with repeated column names will not produce errors.
- Made it so that '.' is removed from the user-input file extension when doing file searching only if the '.' is the first character in the string. This way, file types with multiple extensions, like tar.gz, are now possible to use.
- The raw data generated for Raman was accidently being saved as a csv rather than as a tab-separated txt file.
- Fixed issue when using lmfit.models.ConstantModel for fitting, which gives a single value rather than an array. Now, replace the single value with an array with the same size as the data being fit so that it does not cause errors when plotting.
- Fixed IndexError that occurred when using the fitting GUI and trying to fit residuals.
- Fixed issue where Voigt models with manual peak selection and vary gamma parameter set to True would not set an initial value for gamma.

8.2.3 Other Changes

- Reduced import time of mcetl. On my machine, the import time for version 0.4.0 is ~80% less than version 0.3.0.
- Replaced sympy with asteval for parsing user expressions when creating secondary axes for the plotting GUI. This requires the user to input forward and backward expressions, but otherwise requires no changes. Also, it technically drops a requirement for mcetl, since asteval is already required for lmfit.
- Reordered package layout. Moved all fitting related files to a mcetl.fitting, and moved all plotting related files to mcetl.plotting. This will allow expansion of the fitting and plotting sections without burdening the main folder.
- Renamed peak_fitting_gui to fitting_gui since I intend to extend the fitting beyond just peak fitting.
- Made all of the methods that are only internally used private for DataSource and the Function objects, so that users do not use them.
- Updated required versions for several dependencies.

- Added Python 3.9 to the supported Python versions.
- Created mcetl.fitting.models, which can be filled later with any additional models. Put the modified Breit-Wigner-Fano function in fitting.models.
- Created mcetl.plot_utils that contains all helper functions and classes for plotting.
- The plotting GUI switched back to using "utf-8" encoding when saving data to a csv file (was made to use "raw_unicode_escape" in v0.3.0).

8.2.4 Deprecations/Breaking Changes

- Renamed SeparationFunction to PreprocessFunction to make its usage more clear.
- Changed the file extension for the theme files for the plotting GUI from ".figtheme" to ".figjson" to make it more clear that it is just a json file. Converting existing files should be easy, just change the extension.
- mcetl.launch_peak_fitting_gui() and mcetl.launch_plotting_gui() are no longer valid. Instead, use 'from mcetl import fitting, plotting; fitting.launch_fitting_gui(); plotting.launch_plotting_gui()'.
- The keyword arguments 'excel_writer_formats' and 'figure_rcParams' for DataSource were changed to 'excel_writer_styles' and 'figure_rcparams', respectively.
- DataSource only accepts keyword arguments besides the first argmument, which is the DataSource's name.
- The keyword argument 'peaks_dataframe' for mcetl.fitting.fit_to_excel was changed to 'values_dataframe' to make its usage more clear.
- mcetl.fitting.peak_fitting.fit_peaks no longer takes the keyword 'poly_n' as an argument. Instead, the function takes the keyword 'background_kwargs' which is a dictionary for background keyword arguments, allowing any model to be used as the background. For example, to get the same behavior as with the old 'poly_n' keyword, the new input would be background_kwargs={'degree': 1}.
- Renamed datasource.py to data_source.py. This should have little effect on user code since the DataSource object is available through the main mcetl namespace.
- Renamed the keyword argmument vary_Voigt for mcetl.fitting.peak_fitting.fit_peaks to vary_voigt.
- The constants mcetl.main_gui.SAVE_FOLDER and mcetl.fitting.peak_fitting._PEAK_TRANSFORMS are used instead of the functions mcetl.main_gui.get_save_location (now _get_save_location) and mcetl.fitting.peak_fitting.peak_transformer (now _peak_transformer), respectively. This way, do not need to repeatedly call the functions, and their contents can be alterred by users, if desired.

8.2.5 Documentation/Examples

- Improved the api documentation, added tutorials, and improved the overall documentation.
- Updated example programs for all of the new changes in version 0.4.0.
- Added an example program showing how to use just mcetl.fitting.fit_peaks to do peak fitting instead of using the fitting GUI.
- Changed the readthedocs config to create static htmlzip files in addition to pdf files each time the documentation is built.

8.3 Version 0.3.0 (2020-11-08)

This is a minor version with new features, bug fixes, deprecations, and documentation improvements.

8.3.1 New Features

- Added functions to generate_raw_data.py to create data for pore size analysis (emulating the output of the ImageJ software when analyzing images), uniaxial tensile tests, and rheometry.
- The plotting GUI now uses "raw_unicode_escape" encoding when saving data to a csv file. This has no impact on the data after reloading, but it makes any Unicode more readable in the csv file. The module still uses "utf-8" encoding as the default when loading csv files, but will fall back to "raw_unicode_escape" in the event "utf-8" encoding errors.
- Validation of user-input in the GUIs now converts the string inputs into the desired data type during validation, rather than requiring further processing after validation. Updated all modules for this new change.
- Added the ability to use constraints in the data validation function for user-inputs, allowing user-inputs to be bounded between two values.

8.3.2 Bug Fixes

- Fixed issue where an additional set of data entry column labels was erroneously created when using a SummaryCalculation object for summarizing data for a sample.
- Fixed issue using sorted() with strings rather than integers when sorting the indices of datasets to be deleted when using the plotting GUI.
- Fixed the naming of the standard error for parameters from peak fitting in the output Excel file from "standard deviation" to "standard error".

8.3.3 Other Changes

- The output of the launch_main_gui function is now a single dictionary. This will allow potential changes to the output in later versions to not cause breaking changes.
- The output of launch_main_gui now includes the ExcelWriter object used when saving to Excel. This allows access to the Excel file in Python after running the launch_main_gui function, in case further processing is desired.
- The peak_fitting_gui module now includes full coverage for the data validation of user-inputs for all events.

8.3.4 Deprecations/Breaking Changes

- The output of the launch_main_gui function was changed from a tuple of items to a single, dictionary output.

8.3.5 Documentation/Examples

- Added DataSource objects to the use_main_gui.py example program for the three new raw data types. These analyses are more in-depth than the existing DataSource objects, and involve both CalculationFunction and SummaryFunction objects.
- Changed the Changelog to group changes into categories rather than labelling each change with FEATURE, BUG, etc.

8.4 Version 0.2.0 (2020-10-05)

This is a minor version with new features, bug fixes, deprecations, and documentation improvements.

8.4.1 New Features

- Allow marking and labelling peaks in the plotting GUI.
- File searching is more flexible, allowing for different numbers of samples and files for each dataset.
- The window location for the plotting GUI is maintained when reopening the window.
- The json files (previous_search.json and the figure theme files saved by the plotting GUI) now have indentation, making them more easily read and edited.
- Figure theme files for the plotting GUI now contain a single dictionary with all relevant sections as keys. This allows expanding the data saved to the file in later versions without making breaking changes.
- Allow selecting which characterization techniques are used when generating raw data.

8.4.2 Bug Fixes

- Changed save location for previous_search.json to an OS-dependant location, so that the file is not overwritten when updating the package.
- Allow doing peak fitting without saving to Excel.

8.4.3 Other Changes

- Changed the Excel start row sent to user-defined functions by adding 2 to account for the header and subheader rows. Now formulas can directly use the start row variable, rather than having to manually add 2 each time. Changed the use_main_gui.py example program to reflect this change.

8.4.4 Deprecations/Breaking Changes

- Figure theme (.figtheme) files saved with the plotting GUI in versions < 0.2.0 will not be compatible with versions >= 0.2.0.

8.4.5 Documentation/Examples

- Switched from using plt.pause and a while loop to using plt.show(block=True) to keep the peak_fitting and generate_raw_data example programs running while the plots are open.
- Made all the documentation figures have the same file extension, and made them wider so they look better in the README where their dimensions cannot be modified.

8.5 Version 0.1.2 (2020-09-15)

This is a minor patch with a critical bug fix.

8.5.1 Bug Fixes

- Fixed issue using reversed() with a dictionary causing the plotting GUI to fail with Python 3.7. Used reversed(list(dictionary.keys())) instead.

8.6 Version 0.1.1 (2020-09-14)

This is a minor patch with new features, bug fixes, and documentation improvements.

8.6.1 New Features

- Extended the Unicode conversion to cover any input with `"\"`. This mainly helps with text in the plotting GUI, such as allowing multiline text using `"\n"`, while still giving the correct behavior when using mathtext with Matplotlib.

8.6.2 Bug Fixes

- Fixed how the plotting GUI handles twin axes. Now, the main axis is plotted after the twin axes so that the bounds, tick params, and grid lines work correctly for all axes.
- Fixed an error that occurred when a DataSource object would define Excel plot indices that were larger than the number of imported and calculation columns.
- New DataSource objects that do not provide a unique_variables input will simply have no unique variables, rather than default "x" and "y" variables.
- Fixed an error where column labels were assigned before performing separation functions, which potentially creates labels for less data entries than there actually are.

8.6.3 Documentation/Examples

- Added a more in-depth summary for the package, more explanation on the usage of the package, and screenshots of some of the guis and program outputs to the documentation.
- Added DataSource objects with correct calculations to the example program use_main_gui.py for each of the characterization techniques covered by mcetl's raw_data.generate_raw_data function.

8.7 Version 0.1.0 (2020-09-12)

- First release on PyPI.

CHAPTER

NINE

LICENSE

mcetl is open source and available under the BSD 3-clause license. For more information, refer to the license[439].

[439] https://github.com/derb12/mcetl/tree/main/LICENSE.txt

CHAPTER

TEN

AUTHOR

- Donald Erb <donnie.erb@gmail.com>

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

L

M

N

O

P

R

S

T

V

W

X

Y