Background Reading #4

Table of Contents

1. Package Summary

This document (and the coding exercise) requires the use of several Python packages. These are listed in the table below along with a link to more information and examples for each one.

To install a package, follow the instructions provided in the **Package Import Instructions** file on D2L, found under the Coding Exercises > Download and Setup Instructions folder.

Package or Function Name	Optional Documentation or Useful Guides
csv	The csv module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel
XlsxWriter	XlsxWriter can be used to write text, numbers, formulas and hyperlinks to multiple worksheets and it supports features such as formatting and many more.
Openpyxl	Openpyxl is used to open Excel spreadsheets and extract the containing data.
Pandas	Pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive https://pandas.pydata.org/docs/user_guide/index.html
NumPy	This is a huge fundamental package, so there is no point in providing a link to anything specific. Google is more useful here for individual functions as we come across them.
Matplotlib	https://matplotlib.org/

2. PANDAS

Pandas is a Python package that allows for powerful and flexible open source data analysis and manipulation. The two primary data structures of pandas, series (1-dimensional) and dataframe (2-dimensional), handle the majority of typical use cases in finance, statistics, social science, and many areas of engineering

Pandas is frequently used to take large data sets and quickly sort, order, filter, and perform statistical analysis on them. In nuclear engineering, data could be in the form of instrument measurements (e.g., temperature, pressure, or flow sensor data from various systems) or the output from other codes. Many instruments will take very frequent readings, and while this is useful when analyzing potential upsets to a system or the sensor itself, spreadsheets with thousands of entries are not practical to work with.

Pandas allows you to 'read' an Excel or csv (comma-separated value) file, generate a data structure called a series or dataframe, and then manipulate the data. Some of the power functionality of pandas that we will use in the accompanying Coding Exercise and subsequently in the Assignment include:

- Handling missing data (represented as NaN), whether this is deleting or filling forward/backward (assuming data in row x can be used to fill row x + 1) missing values

- Intelligent label-based slicing**,** indexing**,** and splitting up of large data sets

- Intuitive merging and joining data sets

- Time series-specific functionality: date range generation and frequency conversion, moving window statistics, date shifting, and lagging. Note: we will not have time-series data as our neutron flux profiles will be steady-state, but this functionality is used a lot in engineering.

In the examples below, we will use an existing excel spreadsheet with neutron flux data in D_2O and an infinite slab reactor. This dataset will have entries that are missing, incorrectly set to zero, and at a high frequency (every 1 cm). We will ask pandas to "read" this spreadsheet and to convert it into a dataframe (df) structure. This will allow us to clean up this information into a more useful format (i.e., data wrangling).

2.1 Reading and Cleaning Up Raw Data Files

The function **pandas.read_excel** takes the following arguments:

pandas.read_excel(*io, sheet_name=0, header=0, names=None, …*) (1)

https://pandas.pydata.org/docs/reference/api/pandas.read_excel.html.

We will mainly work with the first two arguments, which are the name of the file and the name of the sheet from which we would like to grab our data. If no identifier is given for the sheet name, the function will take data from the first sheet by default. The header argument uses the entries in the first row as the titles for each column with data by default. These can later be changed to different string labels or to just numbers, if desired. Likewise, by default the "index" or label tagged with each row is an integer count that starts from 0. We can re-index based on one of the rows in our data set (e.g., time, distance, etc). This becomes useful when we are sorting or filtering data, as we can do this based on the independent variable rather than an integer count.

To start, we will use pandas to read the file Flux_InfPlane_CHE5834_D2O_R0.xlsx, which is available under Coding Exercise 4 on D2L. To follow along, you will need to download this file and add it to your Eclipse workspace folder. This allows us to omit the file path when asking Eclipse to look for this file on our computer.

2.1.1 Reading Files with pandas.read()

```
def flux_input_raw():

    # this is the input file name of the doc you have dropped into your workspace
    # the string input provided below must match exactly to the actual file name,
    # including any spaces, dashes, underscores, etc. Easiest is to copy/paste
    # file name directly (e.g., similar to when you go rename a doc, c&p the title)
    Filename = 'Flux_InfPlane_CHE5834_D2O_R0.xlsx'

    InfinitePlaneFlux_df = pd.read_excel(Filename)
    print (InfinitePlaneFlux_df)
```

```
Console Output:

     Distance, x (cm)  Flux (n/cm^2 s)
0                -500     3.218038e+06
1                -499     3.251385e+06
2                -498     3.285078e+06
3                -497     3.319120e+06
4                -496     3.353514e+06
..                ...              ...
995               495     3.388266e+06
996               496     3.353514e+06
997               497     3.319120e+06
998               498     3.285078e+06
999               499     3.251385e+06

[1000 rows x 2 columns]
```

We can preview just a snippet of a dataframe either at the start or at the end using the **.head(n)** or **.tail(n)** functions, where n is the number of rows we wish to see.

```
     print (InfinitePlaneFlux_df.head(5))
```

```
Console Output:

      Distance, x (cm)  Flux (n/cm^2 s)
0              -500     3.218038e+06
1              -499     3.251385e+06
2              -498     3.285078e+06
3              -497     3.319120e+06
4              -496     3.353514e+06
```

2.1.2 Removing Outliers and Missing Values

A plotting function, which can be found in the CodingExercise4_PracticeWorked file, is called to plot flux versus distance. This allows us to see how it behaves and what cleanup is needed. While the shape of the flux is what we expect for an infinite plane, there are missing and negative fluxes, which we will remove. We will also re-index our rows to distances instead of integer numbers.

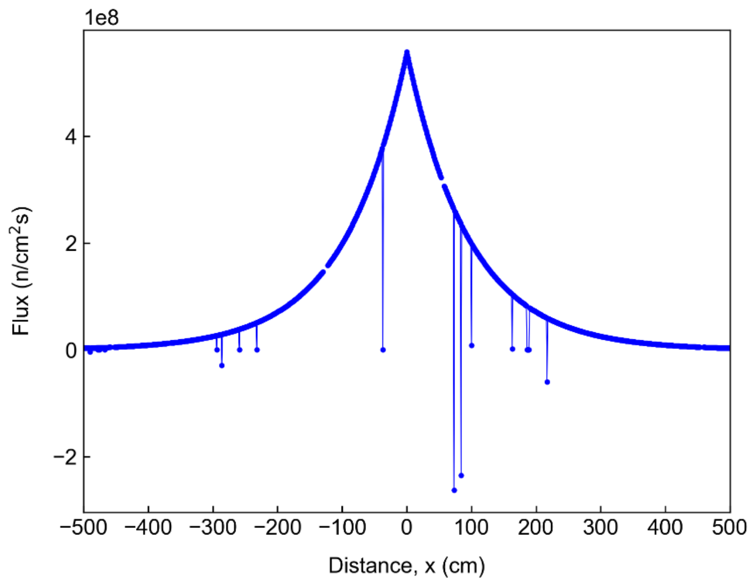

Fig. Unfiltered data for neutron flux in D_2O in an infinite plane.

```
    DistanceLabel = 'Distance, x (cm)'
    Fluxlabel = 'Flux (n/cm^2 s)'

    InfinitePlaneFlux_df.set_index(DistanceLabel, inplace=True)
    Condition = 0
    FluxID = InfinitePlaneFlux_df[FluxLabel]
    # removes 0 values
    Filtered_InfPlnFlux_df = InfinitePlaneFlux_df[FluxID > Condition]
    # drops missing values
    Filtered_InfPlnFlux_df = Filtered_InfPlnFlux_df.dropna(axis=0, how='any')
    print (InfinitePlaneFlux_df.head(3))
```

```
Console Output:

                  Flux (n/cm^2 s)
Distance, x (cm)
-500                 3.218038e+06
-499                 3.251385e+06
-498                 3.285078e+06
```

Replotting with the above filtering results in the following flux. While this is an improvement, we can use additional conditional statements to drop the outlier fluxes.

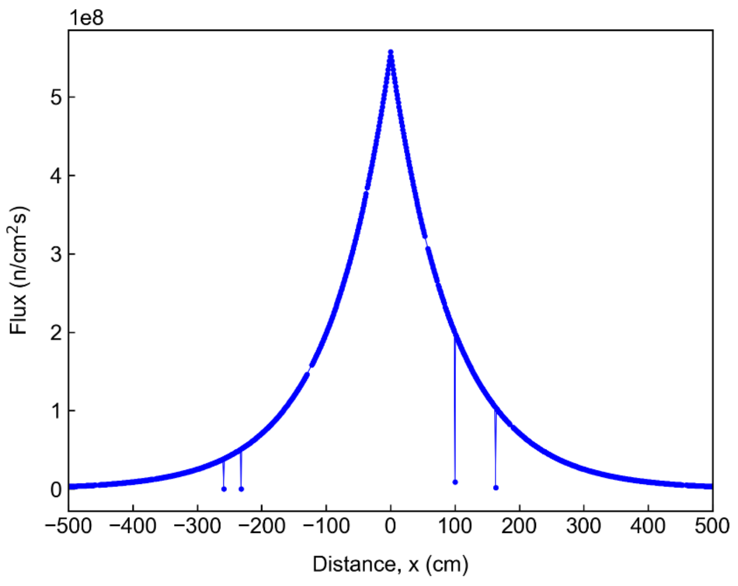

Fig. Filtered data for neutron flux in D_2O in an infinite plane (zero and missing values removed).

The following code tells pandas to remove all rows from the index if the flux is lower than 9 × 10^6 n/cm^2 s at distances between -300 cm and 200 cm. These conditions were chosen to try and remove the low flux outliers from the plot above. The resulting flux dataset, shown below, is much improved from the original.

```
df = Filtered_InfPlnFlux_df
Conditions = (df.index > -300) & (df.index < 200) & (df[FluxLabel] <9e6)
df = df.drop(df[Conditions].index)
```

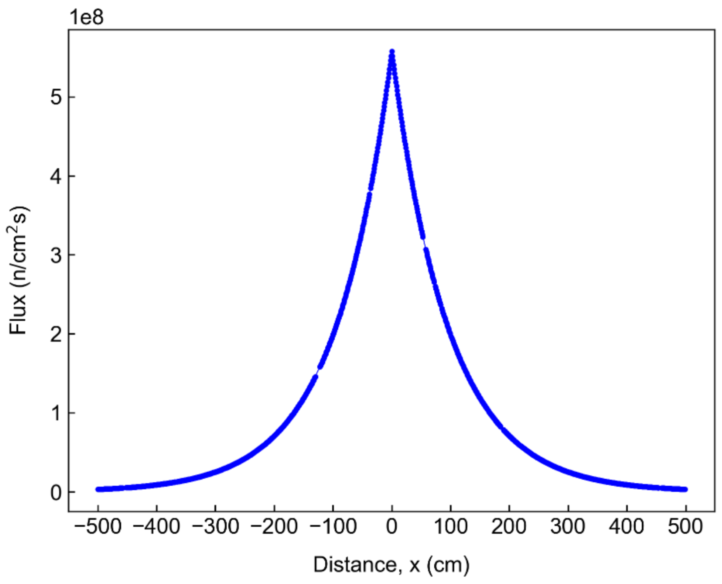

Fig. Filtered data for neutron flux in D_2O in an infinite plane (zero, missing, and outlier values removed).

2.2 Creating a Dataframe from Function Output

The data used to create a dataframe does not have to be in an existing Excel or csv file. This could be generated internally in the code. In our worked example file, the **flux_inf_planar_source()** function calculates and returns the flux of an infinite planar source at the input distance, x. The function **infplaneflux()** then calls on **flux_inf_planar_source()**, visualizes this data by plotting it, then converts the data into a dataframe. The CodingExercise4_PracticeWorked file is by default set to generate the infinite planar flux for neutrons in ordinary (light) water, H_2O. As the diffusion coefficient and diffusion length are different than for D_2O, we expect similar overall shape to the flux but a difference in the magnitude and slope.

```
def flux_inf_planr_source(x):

    ExponentialTerm = np.exp(-abs(x) / DIFFUSION_LENGTH)
    Flux = SOURCE * DIFFUSION_LENGTH * ExponentialTerm / (2 * DIFFUSION_COEFF)

    return Flux


def infplaneflux_calc():

    Distances = []
    LS_Flux_vals = []

    DistanceLabel = 'Distance, x (cm)'
    Fluxlabel = 'Calc Flux (n/cm^2 s)'

    # chose arbitary x value range
    for dist in range (-400, 400, 1):
        Distances.append(dist)
        InfPlnFlx = flux_inf_planr_source(dist)
        LS_Flux_vals.append(InfPlnFlx)


    InfPlFlux_df = pd.DataFrame(
        {
         DistanceLabel: Distances,
         Fluxlabel: LS_Flux_vals
            }
        )

    # plot = flux_plot(InfPlFlux_df)
    # plot.show()

    # plot.savefig('Flux_InfPlane_CHE5834_H2O.png', dpi=300)

    InfPlFlux_df.set_index(DistanceLabel, inplace=True)

    return InfPlFlux_df

infplaneflux()
```

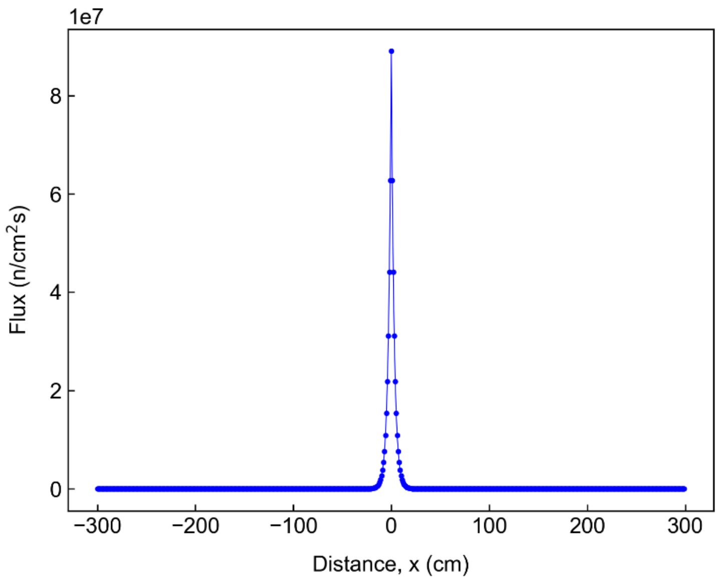

Fig. Function output data for neutron flux in H_2O in an infinite plane.

2.3 Combining Multiple Dataframes

If we wanted to pull a certain column from multiple files (or a combination of files and internal code), then make one new master dataframe with just these columns of data, pandas has a concatenate feature. For instance, we may want to combine the flux from the df we have just cleaned up and a dataframe created from another file, or one generated from a function.

Since we already created the **flux_input_raw()** function to make a dataframe from the Flux_InfPlane_CHE5834_D2OR0 file, we will simply use the returned, already filtered dataframe returned by this function. We will concatenate (combine) this df with a new one that we created using the infplaneflux() function.

The concatenate function has the form:

pandas.concat(*objs, axis=0, join='outer', ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False, sort=False, copy=True*) (2)

https://pandas.pydata.org/docs/reference/api/pandas.concat.html

The **objs** argument is the series of dataframes that we will be combined, while the **axis** argument tells **pandas.concat** which axis we would like to combine along. For example, if we use axis = 0, pandas will add all distance and flux values from the second dataframe to the end of the first one. We can better see this by printing the index from the combined df

```
def combined_flux_input_data():

    A = flux_input_raw()
    B = infplaneflux_calc()

    # axis: 0 = index, 1 = columns, default 0
    AllFluxes = pd.concat([A, B], axis=0)
    for dist in AllFluxes.index:
        print (dist) Console Output:
```

```
Console Output:

496
497
498
499
-400
-399
-398
-397
-396
-395
```

The output is a list from -500 to +499 (range of distances from the first df we combined), with the -300 to + 300 data from the second data frame added to it. However, if we tell pandas to combine with respect to the data columns, not the index, e.g., axis = 1, it will search for flux values in both dataframes at each of the distances from the overall index range of -500 to 500. Since the second dataframe was not evaluated between -500 to -400 or 400 to 500, these values will show up as "NaN".

```
AllFluxes = pd.concat([A, B], axis=1)
# for dist in AllFluxes.index:
#     print (dist)
print (AllFluxes)
```

```
Console Output:

                  Flux (n/cm^2 s)  Calc Flux (n/cm^2 s)
Distance, x (cm)
-500                 3.218038e+06                   NaN
-499                 3.251385e+06                   NaN
-498                 3.285078e+06                   NaN
-497                 3.319120e+06                   NaN
-496                 3.353514e+06                   NaN
...                           ...                   ...
 495                 3.388266e+06                   NaN
 496                 3.353514e+06                   NaN
 497                 3.319120e+06                   NaN
 498                 3.285078e+06                   NaN
 499                 3.251385e+06                   NaN

[986 rows x 2 columns]
```

This is an okay start, but it doesn't combine the data in a useful way. To finish this exercise, we will create a subset of this df over a smaller index range and decrease the index frequency.

2.3.1 Splitting a Dataframe and Additional Filtering

The **.loc()** function will be used to only retain flux values at distances ranging from -200 to 200 cm. Finally, we will change the frequency of the distance index from every 1 cm to every 3 cm.

```
    Flux_df_= AllFluxes.loc[(AllFluxes.index <200) & (AllFluxes.index >=-200)]
```

The **DataFrame.groupby()** function has the following arguments:

DataFrame.groupby(*by=None*, *axis=0*, *level=None*, *as_index=True*, *sort=True*, *group_keys=True*, *squeeze=NoDefault.no_default*, *observed=False*, *dropna=True*) (3)

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.groupby.html

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. The **by** argument is used to determine the groups. This is useful when there are replicate entries to combine. Once the data are grouped, an aggregating function is typically used, e.g., **mean(), sum()**, etc. The following example is from the pandas documentation:

```
df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
                              'Parrot', 'Parrot'],
                   'Max Speed': [380., 370., 24., 26.]})

   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0

df.groupby(['Animal']).mean()

        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

Here, the grouped dataframe consists of averaged speed values for all recurring entries of the same animal. We will use the **numpy.arrange()** function to create an artificial grouping of distances every 3rd element that is the same total length as the number of distance rows in our dataframe. This will be passed for the **by** argument in **DataFrame.groupby()**.

```
    n_rows = 3
    # // operator divides number on its left by the number on its right
    # rounds down the answer and returns a whole number
    rows_for_avg = np.arange(len(Flux_df_)) // n_rows
    print (rows_for_avg)
    Flux_df_ = Flux_df_.groupby(rows_for_avg).mean()
```

```
Console Output:

[0   0   0   1   1   1   2   2   2   3   3   3   4   4   4   5   5   5
   6   6   6   7   7   7 ... 129 129 129 130 130 130 131 131 131 132 132 132 133]
```

3. More on Matplotlib

3.1 DataFrame.plot()

Although we have used Matplotlib previously, using it through pandas will plot all columns against the index without using a loop. To do this, we use the **DataFrame.plot()** function, as shown in the plot function example below.

```
def flux_plot(df):

    df.plot(
        linestyle ='-', marker = '.', markersize = 5, linewidth = 1,
        color = ['blue', 'darkorange']
        )

    plt.xlabel('Distance, x (cm)', labelpad=10)
    # $ signs allow for sub/supersctipt formatting
    plt.ylabel('Flux in an Infinite Plane (n/cm$^2$s)', labelpad=10)
    plt.tick_params(axis="both",direction="in")

    # check how many columns are in the df
    dfnumcols = df.shape[1]

    Label = ['Heavy Water']
    if dfnumcols == 2:
        Label.append('Light Water')

    plt.legend(
        labels = Label, edgecolor='black', loc='upper right', fontsize=11
        )

    # adds label text to plot
    # plot_text(plt)

    plt.savefig('CombinedFlux.png', dpi=300)
    return plt
```

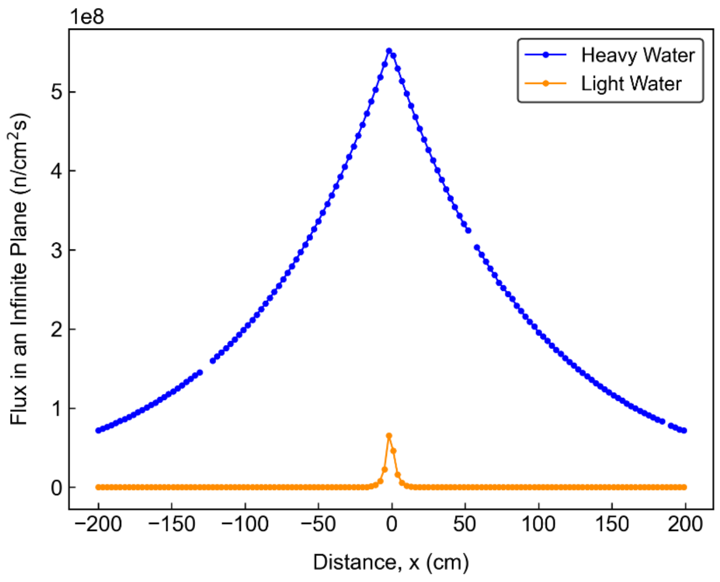

Fig. Filtered dataframe for neutron flux in D_2O in an infinite plane combined with function output dataframe for neutron flux in H_2O in an infinite plane.

3.2 Text Labels and Symbols

In the code above, the superscript is neutrons/cm$^2$ s is formatted by using the `cm$^2$` format. Wrapping ^ value or _ value in dollar signs will result in the formatting of super or sub scripts. The function **matplotlib.pyplot.text()** is used to add custom labels to our plot.

matplotlib.pyplot.text(*x*, *y*, *s*, *fontdict=None*, ***kwargs*) (4)

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.text.html

In the **plot_text()** function shown below, the label "Max source, S (n/cm$^2$s) →" is added to the flux profile for neutrons in D_2O.

```
def plot_text(plt):
    # entire plot is passed as an "object" into this function
    # we will add our desired text to a (x,y) location on the plot object

    # lets us play around with axis specific properties of our plot
    ax = plt.gca()

    # note the use of $^2# for superscript
    label= 'Max source, S (n/cm$^2$s)'
    # could also add break in label using \n, e.g:
    # label= 'Max Source, S \n n/cm$^2$s'

    # method 1: use ax.transdata, which allows us to set location using
    # current plot data values (this is the default, so could omit transform
    # arg from plt.txt below

    # method 1
    x_loc = -82 # cm
    y_loc = 5.67e8 #n/cm^2 s
    trnsfrm=ax.transData

    # method 2 for setting text location:
    # distance can be given in 0 -> 1 location for each axis
    # 0 = origin of x or y plane and 1 = end of plane

    # method 2
    # x_loc = 0.33 # just past middle of x plane
    # y_loc = 0.96 # middle of y plane
    # trnsfrm = ax.transAxes

    # some unicode arrow symbol codes:
    # down: u2193
    # right: u2192
    # left: u2190

    label = label + ' \u2192'
```

```
plt.text(
        x_loc, y_loc, label,
        horizontalalignment='center', verticalalignment='top',
        fontsize='10', color='black', transform=trnsfrm
        )

    return None
```

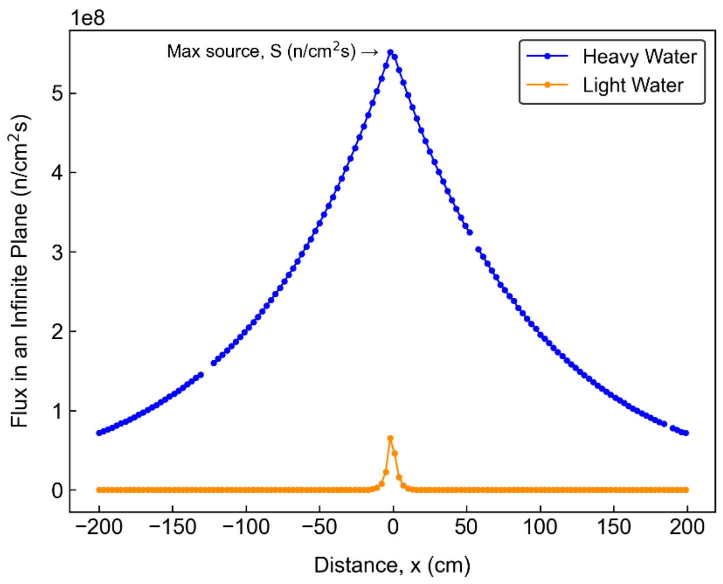

Fig. Filtered dataframe for neutron flux in D_2O in an infinite plane (max flux labelled) combined with function output dataframe for neutron flux in H_2O in an infinite plane.