**PFM: Python Franz Model**

**Basic Overview:**

The model is divided into 4 folders: examples, model, signals, and utilities. This document will exclusively talk about the folders named “model”, “examples, and “utilities”. It will focus on things exclusively pertaining to the Python PFM model, i.e. how to run the Python model. Each function’s inputs, purpose, and what it actually does can be read about in the IDL model.

**General idea of how to run sxr\_x\_emission():**

The Python model was developed to make running PFM easier (okay… I’m totally lying, but at least everything is written in English). The Python model does improve upon some aspects of the IDL model.

First, it is easier to run >1 Line of Sight, especially for the tomography system. The geometric default values used in the Python model are set to those necessary for using the SXR Tomography system. These default values are obvious in the next section below (listed in red in the “Filter” section). As a result, one can easily obtain various plasma parameters for all 80 lines of sight without a large for-loop. In addition, it is faster than the IDL model by ~40% (that was a long time ago. I don’t know what it is now.) How to run this will be addressed more later in the “Specifics for running” section.

Secondly, it is easier to obtain many useful values that are calculated in PFM. These values include all of the spectra, profiles, filter transmission, energy integrated spectra, and detector absorption values. In addition, there are options for finding the line integrated spectra for each contributing factor of the spectrum, i.e. line radiation, bremsstrahlung, etc.

**Default values:**   
Whatever values you do not input when running sxr\_x\_emission() will be set equal to the assigned default values. The Python model’s default values are mostly all the same as the IDL model's, but with a few notable exceptions.  These variables with different default values from the IDL model are written in red in the list below.  The remaining variables with the same default values as the IDL model are written in black. Some variable names were changed from the IDL model. For any variable whose name was changed, the original, IDL name is written in brackets and the variable is color-coded green. Some variables are both green and red.

Temperature  
        te=1250 eV  
        alpha\_te [esp\_te] =4.0  
        beta\_te [esp\_b\_te] =11.0

        del\_Te=0 eV  
        r0=0 (always normalized)  
        del\_r=0  (always normalized)  
        th0=0 rad  
        del\_th=0 rad

Electron Density  
        dens=2.5 (x 1e19 m-3)  
        alpha\_ne [esp\_dens] =10.0  
        beta\_ne [esp\_b\_dens] =1.0

        del\_ne=0.0 (x 1e19 m-3)  
        ne\_r0=0 (always normalized)  
        ne\_del\_r=0  (always normalized)       
        ne\_th0=0 rad  
        ne\_del\_th=0 rad

        dr=0 (always normalized)  
        del\_dr=0  (always normalized)         
        del\_dens=0.0 (x 1e19 m-3)

Neutral Density  
        neutral\_dens=[1.0,100.0,1000.0,0.8] (3 x 1e14 m-3, norm.)

Gaunt Factor  
        gff\_Born\_approx=0.0  
        gff\_E\_Te=0.0  
        gff\_adas=0.0

Geometry and Stuff  
        delta\_h=0 (same units of RADIUS)  
        delta\_a=0 (same units of RADIUS)  
        En0=0 eV  
        nm=0 (%, 1 being 100%)

        n\_qs=5

        n\_s=25  
        radius=1  
        main\_gas='D'

Energy  
        en\_min=20 eV  
        en\_max=20000 eV  
        n\_en=501

Impurities

Note: each of the following must be input as a list whose length is equal to the number length of ‘imp’, in other words, whose length is equal to the number of impurities you desire.

        imp=[] \*Each input impurity must be a string and capitalized in list, i.e. imp = [‘Al’, ‘B’]  
  
        ni\_dens=0.0 (x 1e19 m-3)

        alpha\_ni [ni\_esp\_dens] =0                     
        beta\_ni [ni\_esp\_b\_dens]=1  
  
        del\_ni=0.0 (x 1e19 m-3)  
        ni\_r0=0 (always normalized)  
        ni\_del\_r=0 (always normalized)    
        ni\_th0=0 rad  
        ni\_del\_th=0 rad  
  
        ni\_dr=0 (always normalized)  
        ni\_del\_dr=0 (always normalized)       
        ni\_del\_dens=0.0 (x 1e19 m-3)

Filter/Detector stuff

be\_thick [spessBe] = 80 element array with the first 40 being equal to 857.0um and the last 40 elements being equal to 421.0um.

si\_thick [spessSi] = 80 element array with each element being a relative thickness of a 35.0um detector using the angles of each line of sight on the tomography system

p = 80 element array of angles for each line of sight on the tomography system

phi = 80 element array of angles for each line of sight on the tomography system  
        perc\_comp=0  
        use\_mu=1  
        use\_BESSY=0  
        eff\_mu=0  
        old\_foils=0  
        eff\_filt\_dens=0  
        DSX3\_filt\_dens=0

**Specifics for Running**

There are 2 main functions that you’ll be calling when running PFM. These are: sxr\_x\_emission() and sxr\_line\_integral(). Sxr\_x\_emission() initiates the program, sets all of the default values and other variables, and calculates the filter and detector transmission arrays. Running sxr\_line\_integral() also runs sxr\_spectrum() as a sub-routine. Sxr\_spectrum() calculates all of the various spectra, energy integrated spectra, and profiles for the simulated plasma and outputs all of these values as a large dictionary. Sxr\_line\_integral() uses that dictionary to calculate the line integrated spectra. It proceeds to output the final result: a dictionary containing the line integrated spectra and all of the outputs of sxr\_spectrum(). I will address these functions in order.

sxr\_x\_emission():

sxr\_x\_emission() determines and assigns variables to virtually all of the values used in the other major functions. In addition, it determines the number of lines of sight for the functions to follow. The number of lines of sight is determined by be\_thick, si\_thick, p, and phi. These must all be equal length arrays. Multiple lines of sight can be run by having be\_thick, si\_thick, p, and phi equal to >1 element arrays or you can utilize a for-loop while inputting single elements.

*These are the typical ways to run sxr\_x\_emission():*

1. Default values: st = sxr\_x\_emission()

This calculates the default 80 lines of sight of the tomography system using a plasma with all default values, including no impurities. There are many inputs, but the most important plasma parameter inputs are used in the following example.

1. Important inputs: st = sxr\_x\_emission(te = 1000, dens = 1.0, imp = [‘Al’], ni\_dens = [3.0e-3])

This calculates the plasma parameters for 80 lines of sight of the tomography system using slightly varied plasma parameters. Those which are not specified are set to the default values again. This call to the function changes the core electron temperature, core electron density, impurity list, and core impurity density. Imp must be an empty list or a list containing any number of the following as strings: B, C, N, O, Al. ni\_dens must be an equally long list of core density values corresponding to the specific ions species which are used to calculate the remaining ion densities. These are the same as in PFM.

1. st = sxr\_x\_emission(be\_thick = 2.0, si\_thick = 35.0, p = 1.0, phi = 1.0)

This calculates sets up the plasma parameters for a specific line of sight, namely the filter and detector transmission arrays. All 4 of these parameters can be any sized arrays as well, but they all must be equally sized.

1. st = sxr\_x\_emission(n\_en = 2000)

This calculates the plasma parameters for an energy array of 2000 points. Besides that, it is the same as example 1.

In order to run any of the following functions, sxr\_x\_emission() must be run beforehand because of the class structure of the model.

sxr\_spectrum():

The Python model was designed so that you should run sxr\_x\_emission() followed by sxr\_line\_integral(). It can obviously be run with sxr\_spectrum() in place of sxr\_line\_integral(). You must input correct coordinates though to do this, and you will need sxr\_rho() and sxr\_tzeta() to do this. These two functions translate x and y coordinates to some kind of radial and angular coordinates in MST. You should be able to run all three of these by doing:

*These are the typical ways to run sxr\_spectrum():*

st = sxr\_x\_emission()

rho = st.sxr\_rho(x\_coords, y\_coords)

tzeta = st.sxr\_tzeta(x\_coords, y\_coords)

outputs = st.sxr\_spectrum(tzeta, rho)

The benefits of running sxr\_spectrum() this way as opposed to just running sxr\_line\_integral() [discussed in the next section] is that it is very slightly faster but barely and you have more control over the rho and tzeta coordinates. sxr\_spectrum() by far takes up the most time to run in the entire program. Running sxr\_line\_integral() instead of just sxr\_spectrum() takes virtually the same amount of time.

Secondly, you can assign precise values to rho and tzeta inputs of sxr\_spectrum() much more easily than you could if you ran sxr\_line\_integral(), since sxr\_x\_emission converts x and y coordinates into rho and tzeta coordinates. This is the only reason I can think of for running sxr\_spectrum() instead of sxr\_line\_integral.

The output of sxr\_spectrum is a dictionary with three sub-dictionaries: ‘profiles’, ‘spectra’, and ‘en\_int’.

profiles:

nip: ion profiles for each impurity and deuterium

ions: list of ions in the plasma. The order of the ions in this list is the same as the profiles in nip

nep: electron density profile

tep: temperature density profile

nHp: neutral density profile

gff: free-free gaunt factor

gfb: free-bound gaunt factor

en\_integral:

en\_int\_brems: energy integrated bremsstrahlung spectrum

en\_int\_b\_imp:

en\_int\_recomb: energy integrated recombination spectrum

en\_int\_r\_imp:

en\_int\_lines: energy integrated aluminum line radiation spectrum

en\_int\_lines\_11: energy integrated aluminum 11+ line radiation spectrum

en\_int\_lines\_12: energy integrated aluminum 12+ line radiation spectrum

en\_int: energy integrated total spectrum

en\_int\_b\_r: energy integrated bremsstrahlung and recombination spectrum

spectrum:

sp\_brems: spectrum from bremsstrahlung

sp\_b\_BeSi: spectrum from recombination multiplied by the filter transmission and detector absorption

sp\_b\_imp: spectrum from bremsstrahlung of each impurity

sp\_b\_i\_BeSi: spectrum from bremsstrahlung of each impurity multiplied by the filter transmission and detector absorption

sp\_recomb: spectrum from recombination

sp\_r\_BeSi: spectrum from recombination multiplied by the filter transmission and detector absorption

sp\_r\_imp: spectrum from recombination of each impurity

sp\_r\_i\_BeSi: spectrum from recombination of each impurity multiplied by the filter transmission and detector absorption

sp\_lines: total spectrum from aluminum lines in units of W/m^3/eV

spectrum: total spectrum in units of W/m^3/eV

energy: array of energies calculated in sxr\_x\_emission()

For example, to get these values, you can run what we did before at the beginning of the sxr\_spectrum() section where we have outputs = st.sxr\_spectrum(tzeta, rho). Now, for example, to get energy, we have: outputs[‘spectrum’][‘energy’]. We’ll see this come into play in the next section as well when we talk about sxr\_line\_integral().

sxr\_line\_integral()

To start, sxr\_line\_integral outputs everything as a big ass dictionary, henceforth named the Dictionary of Big Assness (DOBA). In DOBA, there are two sub-dictionaries: ‘sxr\_spectrum’ and ‘sxr\_line\_integral’. ‘sxr\_spectrum’ is a dictionary of values corresponding to all of the outputs of sxr\_spectrum(). The ‘sxr\_spectrum’ sub-dictionary has tags: ‘profiles’, and ‘en\_integral’, and ‘spectrum’. The variables are defined in the sxr\_spectrum() section above.

sxr\_line\_integral() runs sxr\_spectrum() to calculate all of its outputs then runs a simple line integral over the line of sight’s length to get the W/m^2 signal. IDL PFM does this for the total SXR radiation only. The Python model can do this for every source of radiation. There are just some simple inputs you can set to true (1) when running sxr\_line\_integral(). These values are all set to false (0) by default. The possible, optional inputs and their default values are the following: signal\_lines=0, signal\_brems=0, signal\_recomb=0, signal\_lines\_11=0, signal\_lines\_12=0. Most of these are obvious, but signal\_lines\_11=1 calculates the signal for Al+11 and signal\_lines\_12=1 does it for Al+12. The signal array outputs will be arrays equal in size to the number of line of sights specified when initializing sxr\_x\_emission().

*These are some sample ways to run sxr\_line\_integral:*

1. outputs = st.sxr\_line\_integral()

This calculates the line integrated spectrum for the TOTAL spectrum only. It does it for each line of sight, so you end up with an array with an equal number of elements as the line of sights that were set when running sxr\_x\_emission() before this. Again, it outputs all of the outputs from sxr\_spectrum() and sets those as values in the output dictionary.

1. outputs = st.sxr\_line\_integral(signal\_lines = 1, signal\_brems=1, = 1, signal\_recomb = 1, signal\_lines\_11 = 1, signal\_lines\_12 = 1)

This calculates the line integrated spectra (the signal) for ALL components of the total spectrum in addition to the total spectrum itself. These are values that are output in the sub-dictionary ‘sxr\_line\_integral’.

For example, to get the line radiation signal, you’d do this: outputs[‘sxr\_line\_integral’][‘it\_lines’]. To get bremsstrahlung, you’d do this: outputs[‘sxr\_line\_integral’][‘it\_brems’].

And, to get the output from sxr\_spectrum, you’d do this: outputs[‘sxr\_spectrum’] Here you can use any other tag following [‘sxr\_spectrum’] in order to obtain the output arrays from sxr\_spectrum() just like you ran sxr\_spectrum() by itself. For example, to get energy this way, as we did in the sxr\_spectrum() section, we do: outputs[‘sxr\_spectrum’][‘spectrum’][‘energy’].

**Example Program:**

The example program I am talking about is model\_example.py located in the Examples folder. This is just a program to familiarize directly with how the model is generally ran and how some functions can be called, namely the sub-routines talked about below. I will not discuss it more than this.

**Sub-routines**

These are located in the ‘Utilities’ folder. Three of these files house different functions that I have used when designing the NICKAL2 detector, namely those for transmission plotting of filters and passbands as a function of energy, signal calculations for a single line of sight, plotting signal for single or multiple filters or passbands vs Te(0) or ne(0), and, finally, plotting spectra vs. E (generally with transmission as well). These three files are: measure\_filter\_signal.py, optimization\_routine.py, and plot\_functions.py. I do not discuss how to run these programs. I believe that I have documented these well in the files themselves, I have examples calling all of the useful functions in model\_example.py at the bottom. I also have a second example for calling measure\_signal\_vs\_x() commented out at the bottom of measure\_filter\_signal.py.

measure\_filter\_signal.py:

measure\_filter\_signal: what this function does is measures the signal for a single filter of at least 1 and up to 3 elements.

measure\_passband\_signal: what this function does is measures the passband signal for at least 2 filters and up to 3 filters. This means there can be either 1 passband if you use 2 filters or 3 passbands if you use 3 filters.

measure\_2\_passband\_signal: what this function does is measures the various components of the signal from each of the three filters required for this function. See measure\_filter\_signal.py to become familiar with inputs and outputs. These outputs are used in various plotting functions located in plot\_functions.py.

measure\_2\_element\_filter\_signal: what this function does is measures the various components of the signal from a single filter that is composed of 2 elements. This is a specific case of measure\_filter\_signal(). This is only here because it is used within measure\_signal\_vs\_x() and I do not have time to change measure\_signal\_vs\_x() to utilize the newly made: measure\_filter\_signal() function. DO NOT USE THIS CODE OTHERWISE. Measure\_filter\_signal() is strictly more versatile.

measure\_signal\_vs\_x: what this function does is measures the various components of signal for three filters at a range of x, where x is specified as either core electron temperature or core electron density. This function requires 3 filters and it requires 2 elements in each filter. That said, you can set the percent composition of your first element to 100.0 so the second element doesn’t contribute to the transmission. Sadly, I do not have time to make this more general.

signal\_noise: calculates the noise for an array depending on whether the array is a ratio of signals (4% noise) or just an array of signals (4% noise for each signal added in quadrature).

optimization\_routine.py:

global\_optimization: given a range of thicknesses and percent compositions of three filters, determines the filter combination that gives the maximum aluminum line radiation signal within the passbands. Utilizes the figure\_of\_merit() function discussed below.

Figure\_of\_merit: This is the function that calculates the figure of merit F­I discussed in my APS poster. It does it for a single filter combination. Must be looped over for the optimization routine.

Passband\_optimization: This is another way optimizing the passbands. It’s the same as global\_optimization but uses only a single function from scipy.

plot\_functions.py:

measure\_emiss: calculates the spectra, energy array, and transmission curves of 3 filters each with a layer of Be. DO NOT USE THIS FUNCTION. It is called in plot\_NICKAL2\_transmission(), which is the only reason it’s in the file still. See plot\_filter\_transmission() for the comparison.

plot\_filter\_transmission: calculates and plots the transmission of a filter composed of at least 1 and up to 3 elements. Energy is on x-axis and limits can be specified on input. Whether the spectrum is also plotted can be specified by including ‘spectrum = 1’ at the end of the inputs. This will plot the spectrum of the plasma for a single line of sight with otherwise default values along with the transmission curves.

plot\_NICKAL2\_transmission: used with measure\_emiss. Plots the SXR spectrum and transmission arrays vs energy for the three elements used in the NICKAL2 detector [‘Zr’, ‘Al’, and ‘Si’]. The default values are the actual values for the filters used in NICKAL2.

plot\_passband\_transmission: using similar inputs to measure\_passband\_signal(), it plots the passbands’ transmission as a function of energy for at least 2 and up to 3 filters. Spectrum can be plotted as well.

plot\_signals\_vs\_x: used in conjunction with measure\_signal\_vs\_x. Plots passband signal vs. core electron temperature or core electron density. It plots signal with error, and it plots all types of signal included in the input array, which is also the output from measure\_signal\_vs\_x(), vs the specified x.

**Questions**

If you have any questions, try asking Patrick or Lisa. Otherwise, you can always email me at: [njlauersdorf@wisc.edu](mailto:njlauersdorf@wisc.edu) and I can try to answer them if they can’t.