HOW TO:
USE THE ORBIT WEIGHT COMPUTATIONAL FRAMEWORK (OWCF)

BY HENRIK JÄRLEBLAD

HOW TO:

USE THE ORBIT WEIGHT COMPUTATIONAL FRAMEWORK (OWCF)

BY HENRIK JÄRLEBLAD

Welcome! Thank you for using the orbit weight computational framework (OWCF). It is a code framework written in the Julia programming language. It is currently fully integrated with the DRESS code written in Python, a code framework for computing energy spectra produced by (one-step) fusion reactions. The DRESS code was written by J. Eriksson and others. The OWCF is modular in the sense that any other forward model can be coupled to the OWCF, to compute orbit weight functions, such as FIDASIM.

When using this guide, please remember that it could be outdated and you might need to find an updated version.

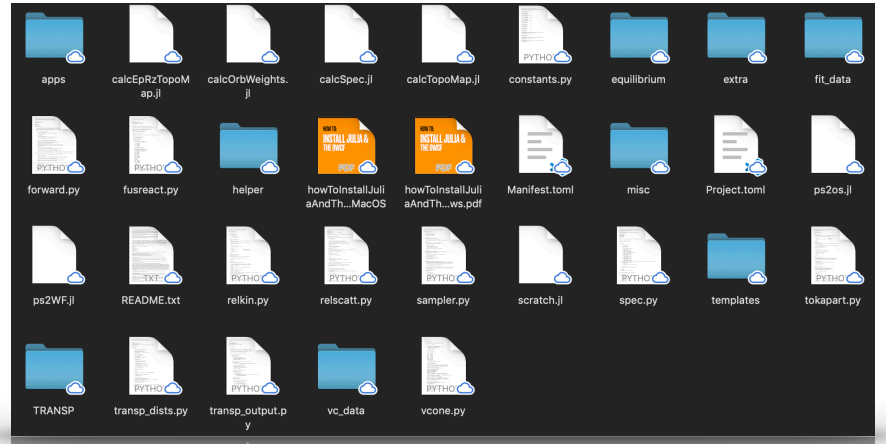
This version was written in October of 2022.

Good luck with using the OWCF!

Henrik Järleblad
Plasma physics and fusion energy
DTU Physics
Technical University of Denmark
www.henrikjarleblad.se

1.
AN OVERVIEW

The OWCF consists of many scripts that form both an integrated whole, but they could also be used independently, should that be desired. Below you find a screenshot of all the scripts and folders as of October 2022. What might be important to notice, is that there are both Julia files (with extension .jl) and Python files (with extension .py). The Python files originate from the DRESS code, while the Julia files are inherent to the OWCF. You also



Screenshot of the files and folder in the OWCF, as displayed on a Mac. Please ignore the small cloud icons.

have a README.txt file. Read this if you want a compressed version of everything in this guide.

It is important to note that, as of October 2022, the OWCF-DRESS code coupling can only compute neutron emission spectroscopy (NES) orbit weight functions and gamma-ray spectroscopy (GRS) orbit weight functions. I will tell you what specific fusion reactions that are included later in the guide (you can also check the misc/availReacts.jl file as well as the start files of the calcOrbWeights.jl and calcSpec.jl scripts). For fast-ion D-alpha (FIDA) spectroscopy orbit weight functions, you can use the FIDASIM code, available for download via the FIDASIM webpage. A guide on how to use the OWCF together with FIDASIM will be included in a future version of this manual.

In addition, for this guide, I will assume that you have correctly installed Julia and downloaded, and pre-compiled, all the necessary packages. Please refer to the *howToInstallJuliaAndTheOWCF_Windows.pdf* (or *howToInstallJuliaAndTheOWCF_MacOS.pdf*) document for instructions on how to do that. It's available in the OWCF/ folder.

Now, let's start with the question that you might be asking yourself right now. How do I compute an orbit weight function? Well, let's start from the beginning.

2.
COMPUTING ORBIT WEIGHT FUNCTIONS

- calcOrbWeights.jl

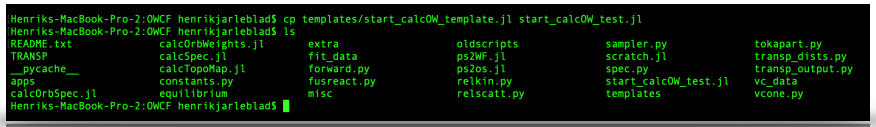
In practice, most computations done with the OWCF are done via submit files and start files. The submit files are only necessary to use if you are working on a computational cluster. The templates for the submit files are written so as to work on computational clusters using the SLURM workload manager (such as on the DTU Niflheim supercomputer cluster). If your computational cluster does not use the SLURM workload manager, you have to write your own submit files (for e.g. PBS clusters).

I will show you how to use the submit files in chapter 9. They will be necessary if you want to compute quantities with an orbit-space grid resolution higher than, let's say, 50x100x100. For now, let's start with only the start files.

The first step is to copy the *start_calcOW_template.jl* from the templates folder into the OWCF folder. You will find the templates/ folder in the OWCF folder. Throughout this guide, I will be using the terminal to showcase the process steps. As you can learn by reading the *howToInstallJuliaAndTheOWCF.pdf* documents, it is a very convenient tool once you learn how to use it. It is also the place where you will be executing all your OWCF computations. So go ahead, open a new terminal window. On Mac, go to the Desktop and press 'cmd+spacebar' and type

'terminal'. Then hit enter. A new terminal window will open. On Windows, go to the search function and type 'powershell'. Then hit enter. A new terminal window (powershell) will open. Then, using the 'cd' command (same on Mac and Windows), navigate to the OWCF folder (you have to remember where you saved it. How to navigate with the help of the terminal is explained more in detail in the *howToInstallJuliaAndTheOWCF.pdf* documents).

Now that you are in the OWCF folder, go head and copy the *start_calcOW_template.jl.* It is a good idea to rename the file as you copy it. Do like the picture below shows.



Use the 'cp' command to copy. On Windows, that's 'copy' instead of 'cp'. Also, on Windows, the 'ls' command is 'DIR', which let's you see the content of a folder

Notice how the start file template has been copied into the OWCF folder. Now, you will use a code editor of your choice (Visual studio code, IntelliJ, notepad etc) to edit the start file and prepare it for execution. Let's take a look at the start file and see what it contains.

The first thing you will notice as you open the file is that there is a long description at the top. This is common for most scripts in the OWCF. Here you will find a description of the script, important information as well as a breakdown of all the start inputs (input variables) that you can specify, and what their Julia types are (Int64, String, Bool etc). Now, scroll down to the actual input variables.

The first input variables you will see are the system specification variables. These will govern how the computation is executed. The variables are as follows

- **batch_job** - If true, then the script will assume that it's being executed via a batch job on a SLURM computational cluster (via the SLURM command 'sbatch' and a .sh file, for example). Set this variable to false if this is not the case.

```
## First you have to set the system specifications
using Distributed # Needed, even though distributed might
batch_job = true
distributed = true
numOcores = 40 # When executing script via HPC cluster job
```

The computational input variables

- **distributed** - If true, then multi-core computation will be used. This is generally preferable. However, if you are searching for bugs, or if you are computing orbit weight functions with a really small orbit grid, then it might be a good idea to set the 'distributed' variable to false.
- **numOcores** - An integer (in Julia, integers have no decimal part). This will specify the number of CPU cores to be used for certain computations, regardless if batch_job is set to true or not. PLEASE NOTE! If batch_job is set to true, this number must match the number of cores that you have specified with the '-n' flag in your .sh shell script file (please see chapter 9 for more info).

Then, as you scroll down the start file, you will see all the actual input variables. I will not go through them all here. For info, I invite you to read the description at the top of the start file, as well as the short description written as comments to the right of the input variables. I will just mention that

- **analyticalOWs** should be set to false, unless you would like to compute analytical orbit weight functions. That is, orbit weight functions where the projected velocity of the fast ion is computed for all points along the orbits. These projected velocities are then binned into an 'analytical'

spectrum. No thermal plasma profiles are needed. The projected velocities are the projections of the ion velocity vectors onto the diagnostic line-of-sight.

- The **filepath_equil** input variable must be the path to a file containing magnetic equilibrium data. It can be either a Solov'ev equilibrium output file, computed using the *misc/compSolovev.jl* script. Or it can be an **.eqdsk/.geqdsk file**. An .eqdsk file is a text file (but it has the file extension .eqdsk or .geqdsk) containing information about the magnetic equilibrium for a specific TRANSP ID. These are generated automatically when running TRANSP. For a complete breakdown of what .eqdsk files contain, please see the misc/eqdsk_file_breakdown.pdf document.
- **inclPrideRockOrbits** is a boolean (true/false) variable. Its name is a reference to the discovery of the region of orbits whose R_m coordinates are smaller than the magnetic axis (high-field side stagnation/counter-stagnation orbits basically). Back then, the shape of the counter-stagnation orbit region in (E, p_m, R_m) orbit space was not known. When it was discovered, the author thought that its shape was very similar to that of Pride Rock in the classic Disney movie 'The Lion King'. And that's how the name 'Pride Rock orbits' came to be.
- **include2Dto4D** should be set to true, if the 2D orbit weight matrix should be inflated into its true 4D form upon completion of computations. This is the case if the orbit weight functions are to be analysed with the weightsWebApp.jl, for example.
- **filepath_thermal_distr** is the path specification to a file of either .cdf or .jld2 format (or simply leave it unspecified). The .cdf file is obtained from the TRANSP shot and, for JET, can be found on the UKAEA Heimdal/Freia computational clusters for example (at '/common/transp_shared/Data/result/JET/'). filepath_thermal_distr can also be specified as a .jld2 file. Please see the calcOrbWeights.jl script file for more info. Finally,

filepath_thermal_distr can also be left unspecified (“”). A default temperature and density profile will then be used. These default profiles are fitted functions from the temperature and density profiles of JET shot 96100, and can be viewed in misc/dens_fit.png and misc/temp_fit.png

- **filepath_FI_cdf** is the path specification to a (.cdf) TRANSP fast-ion shot file that matches the (.cdf) TRANSP shot file that might have been specified with the filepath_thermal_distr variable. You have to specify a TRANSP *fast-ion* shot file if you want to use a (.cdf) TRANSP shot file for the filepath_thermal_distr variable. This is because the TRANSP fast-ion shot file contains information about which time window of the shot file to use. Please note! You can leave the filepath_FI_cdf variable unspecified if you specify a .jld2 file for the filepath_thermal_distr variable, or if you leave the filepath_thermal_distr variable unspecified.
- **visualizeProgress** should be set to false if you are executing the script via a computational cluster batch job

Then you also have the EXTRA KEYWORD ARGUMENTS. These can be created and specified yourself. However, you have to put them into the **extra_kw_args** dictionary, as shown in the template file. Extra keyword arguments that are pre-specified by default are:

- **toa** stands for ‘try only adaptive’. If true, then the algorithm that performs the integration of the equations of motion for the guiding-centre will only try adaptive integration (the size of the time step changes dynamically in response to the stiffness of the integration). If false, then if the algorithm fails in trying adaptive integration, it will also try non-adaptive integration. Non-adaptive integration is essentially when the size of the time step stays constant throughout the integration. In certain situations, this might be good to try if the adaptive integration fails. However, as we’ll see in the *maxiter* extra keyword argument, that

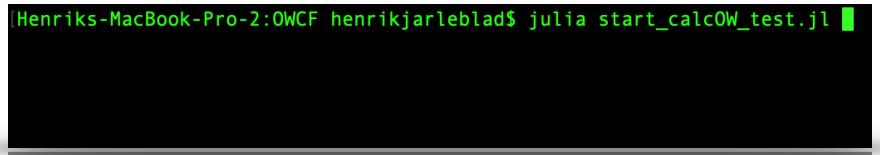
might result in the algorithm stalling and the integration failing while using up increasing amounts of RAM.

- **limit_phi** is an extra keyword argument that limits the guiding-centre trajectory in the toroidal direction. If the particle has travelled more than *maxphi* radians in the toroidal direction (*maxphi* is set to 10 toroidal turns (2*pi*10) by default) without completing even one orbit in the poloidal direction, then the integration is terminated. Setting this extra keyword argument to true will prevent stalling, but might result in a small number of orbits being labelled wrongly as 'incomplete' orbits.
- **maxiter** is an extra keyword argument that limits the number of times that the algorithm tries smaller and smaller time steps. This only becomes important if the algorithm failed to integrate the equations of motions via adaptive integration, and the extra keyword argument *toa* is set to false. Every time the algorithm tries smaller time steps, it divides the size of the current time step by a factor of 10. Thus, if the integration is really stiff than there is an increasingly larger chance that the integration will succeed by using a smaller time step. However, a larger and larger amount of RAM will then be required. Thus, setting the extra keyword argument *maxiter* to a low integer (0 or 1) will ensure that the algorithm will not explode in terms of RAM usage. It will also ensure that the algorithm returns an 'incomplete' orbit rather than performs increasingly longer integration attempts.

In the appendix, a justification is given in favour of returning 'incomplete' orbits rather than using non-adaptive integration and smaller and smaller time steps.

When you have specified all the input arguments in your start file it is time to start the computation of the orbit weight functions (the orbit weight matrix). You start the computation by navigating to the OWCF folder (where you have saved your start file) in your terminal (or powershell) and starting the

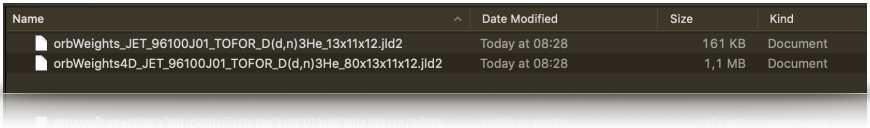
computation by typing ‘julia start_calcOW_test.jl’ in your terminal (or powershell). Like the screenshot below shows.



Type ‘julia start_calcOW_test.jl’ and hit enter to start the computation

Of course, your file does not have to be named ‘start_calcOW_test.jl’. In fact, it should not be. You should always replace the ‘test’ or ‘template’ part with a unique identifier. This is to avoid confusion for yourself, give structure to your work and avoid accidentally overwriting stuff.

If all goes well, you should have your orbit weight functions in the *folderpath_o* folder that you specified in the start file. If you also specified *include2Dto4D* as true, you should also have a .jld2 file with the orbit weight functions in 4D format (with dimensions (E_d, E, p_m, R_m)). Like the example screenshot below shows.



Results of the calcOrbWeights.jl script. The actual filenames might contain more/less information than the examples in the figure

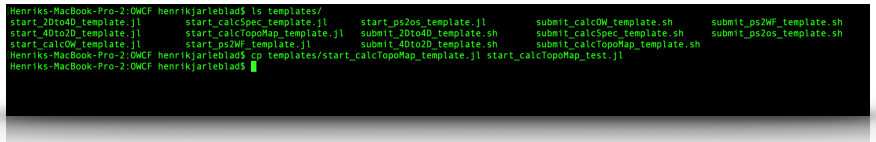
Notice how the 4D version of the orbit weight matrix takes up much more space. Therefore, only compute this 4D quantity if you really need it (e.g. for visualisation purposes). This concludes the chapter on how to compute orbit weight functions.

3.
COMPUTING ORBIT-SPACE TOPOLOGICAL MAPS

- calcTopoMap.jl

Before we can visualise orbit weight functions, we are going to need to compute orbit-space topological maps to be able to navigate orbit space. These topological maps will also serve as the basis with which to compute orbit-space topological boundaries, so that we can differentiate between the topological regions of orbit-space (co-passing, trapped, potato etc) when visualising the orbit weight functions. Let's begin.

You start by doing as you did in the previous chapter. You copy a start file from the OWCF/templates/ folder to the main OWCF/ folder. As the screenshot below shows (notice how I change the start-file identifier from 'template' to 'test' in the process).



Copy calcTopoMap start file from OWCF/templates/ to / templates/ and change name in the process

As with the calcOrbWeights.jl start file, there will be input variables that need to be specified. Let's take a look at them.

First, at the top, we have a start-file description and the system specification variables, just like before.

Then, as we scroll down, we will see the familiar filepath_equil input variable. But then we have new friends to get to know. Some of them include (**please note, not all input variables need to be specified**):

- **filepath_W -** This is the path to the orbit weight functions file to extract required orbit-grid and particle species information from. Can be either an orbWeights_...jld2 or orbWeights4D...jld2 file.
- **filepath_OG** - This is the path to the orbit grid file to extract required orbit-grid and particle species information from. Should be an output file from calcOrbGrid.jl.
- **weightsFileJLD2** - If false, it will assume that the orbit weights file is in .hdf5 format
- **keyname_diagHDF5** - There are certain old orbit weight functions files that require the diagnostic identifier to be specified to be able to load. For example 'TOFOR' or 'afterburner'
- **saveTransitTimeMaps** - If set to true, the poloidal and toroidal transit times for all valid orbits will be saved as well. These can then be visualised via e.g. the orbitsWebApp.jl (later chapter).
- **distinguishLost -** Explained in start file and calcTopoMap.jl script.
- **distinguishIncomplete** - Explained in start file and calcTopoMap.jl script.
- **includeExtractTopoBounds** - If set to true, then the topological boundaries will be computed from the topological map, and extracted to a .jld2 file which will be saved in the same output folder. This quantity is what you will need for the weightsWebApp.jl for example (given as input variable filename_tb. Please see chapter 4)

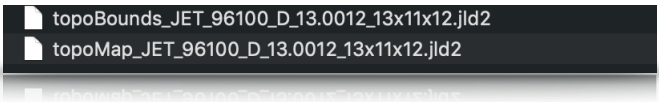
Then there are further input variables that are necessary to specify if you do not want to base your topological map on the information found in an orbit weights functions file (i.e. the

```
## ------------------------------------------------------------------------
@everywhere begin
    distinguishIncomplete = false
    distinguishLost = false
    filepath_equil = ""
    filepath_OG = ""
    filepath_W = ""
    folderpath_OWCF = ""
    folderpath_o = "../OWCF_results/template/" # Output folder path. Finish with '/'
    includeExtractTopoBounds = false # If true, then a .jld2 file with the boundaries of the topological map
    keyname_diagHDF5 = "" # Needed to extract info from .hdf5-file, if weightsFileJLD2 is set to false
    timestamp = 00.0000 # If unknown, just set 00.0000
    tokamak = "" # If unknown, just set ""
    TRANSP_id = "" # If unknown, just set ""
    useOrbGridFile = false # Set to true, if you want to load the orbit grid from the orbit grid file filepat
    useWeightsFile = false # Set to true, if you want to load the orbit grid from the orbit weights file file
    verbose = true # If set to true, the script will be very extroverted! Yay!
    visualizeProgress = false # If set to true, a progress bar will be displayed during distributed computati
    weightsFileJLD2 = false # Set to true, if useWeightsFile is set to true, and filepath_W is .jld2 in file
```

The first set of input variables for calcTopoMap.jl, after the system specification variables

input variable ‘useWeightsFile’ is set to false). These should be pretty self-explanatory. Please see the start file and calcTopoMap.jl for further info.

Now, if you have specified your start file variables correctly, please go ahead and save the file. Then, just as before, go out to the terminal, navigate to the OWCF folder and type ‘julia start_calcTopoMap_test.jl’ to execute the script. Then, the script will execute. When it is done, you should see a file (or two, depending if you set *includeExtractTopoBounds* to true or not) in the *folderpath_o*



Output files of the calcTopoMap.jl script.

folder named something like the figure above shows.

The pattern for output files is often (or similar):

‘[type]_[tokamak]_[TRANSP ID]_[species/reaction]_[timestamp]_[orbit-grid size]’

which is exactly what we see in the figure. Now, we are ready to visualise our weight functions with the help of our topological boundaries!

4.

VISUALISING ORBIT WEIGHT FUNCTIONS

- apps/weightsWebApp.jl

The OWCF contains several apps that can be used to visualise orbit weight functions in an interactive and intuitive manner. Using the apps, one can also easily save the plots as .png files. Visualising orbit weight functions in an interactive manner is often the best way of doing it, since they are 3D quantities and interacting with the data gives a better sense of understanding than static plots can achieve.

The first app that we are going to make ourselves familiar with is the **weightsWebApp.jl**. This app has many features, and some of them we will return to and revisit in later chapters. For now, let's just use weightsWebApp.jl to visualise the orbit weight functions we computed in chapter 2.

As with all other scripts in the OWCF, we must start by specifying some input variables. The difference here is that we are going to modify the script/app directly, without using a start file or similar. Let's take a look.

Navigate to the OWCF/apps/ folder. In here, you should see a file named weightsWebApp.jl (please notice the 's' in weights!!!). Open it using a code editor of your choice (Visual studio code, IntelliJ, notepad etc). Please read the file description before continuing. Then, once you're done, go ahead and scroll down to the input variables. Let's take a look at them and see what we need to specify.

- **port** - This is the internet I/O port that your computer is going to use to host the app. Don't worry, it is not going to access the internet. It simply needs an I/O port to allow

you to access it via a web browser of your choice (internet explorer, Google chrome, Firefox etc). However, as is discussed in the appendix, you can access the OWCF apps via the internet, from another computer, if you really want to. Please go to the appendix for more info. For now, just specify the *port* input variable to an integer that you know that your computer doesn't use for any other active I/O on your computer (the OWCF default for weightsWebApp.jl is 9999).

- **verbose** - If true, the app will talk a lot!
- **filepath_tb** - The path to the topological boundaries .jld2 file (topoBounds_... etc) that you computed in chapter 3.
- **enable_COM** - If true, the weightsWebApp will allow you to view the weight functions in both (E,pm,Rm) and (E,mu,Pphi;sigma) coordinates. You switch coordinate system via a toggle button in the web application.
- **filepath_W_COM/filepath_tm** - If enable_COM has been set to true, I highly recommend you to specify either a filepath to the orbit weight functions already in (E,mu,Pphi;sigma) format (can be an output file from e.g. the helper/os2com.jl script, or saved automatically be the weightsWebApp if neither filepath_W_COM nor filepath_tm was specified), or to specify the filepath to a topological map. This will help weightsWebApp make the mapping from (E,pm,Rm) to (E,mu,Pphi;sigma) as smooth and efficient as possible.
- **filepath_equil** - This one we know what it is now!
- **diagnostic_filepath** - The filepath to the LINE21 output file that contains the R, z, x and y (and more) data for the diagnostic viewing cone. You can also leave it as "". Then, no viewing cone will be plotted by the app.
- **diagnostic_name** - The name of the diagnostic of the viewing cone your want to visualise. This input variable is purely for aesthetic purposes. If you specify "TOFOR" for example, the weightsWebApp.jl will choose the default

OWCF TOFOR-color, which is green, to plot the viewing cone.

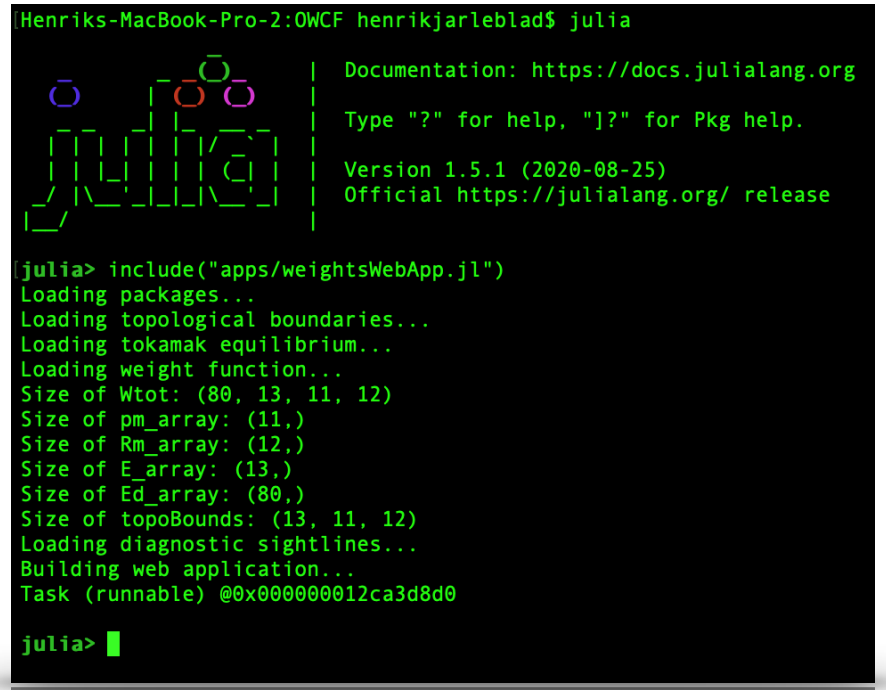
- **filepath_W** - The path to the 4D orbit weight matrix file that you want to visualise.
- **weightsFileJLD2** - Should be set to true if the orbit weights have been computed with the OWCF anytime after July 2021. The app will assume the weights file are in .jld2 format.

Then, we have some input variables that we will save for later chapters. I will just mention that

- **FI_species** - The particle species of the fast-ion orbits that you want to visualise. The app will try to figure out the particle species from the orbit weights file. But if it doesn't succeed, it will need a backup. Please see the misc/species_func.jl for available fast-ion species to choose from.
- **folderpath_OWCF** - The path to the OWCF folder. Remember to always finish folder paths with "/".

Now we are ready to start the app. To do so, navigate to the OWCF folder in your Mac terminal/Windows powershell, and start Julia by typing 'Julia' and hit enter. Julia will start and then you should type 'include("apps/weightsWebApp.jl")' and hit enter. The app will start to boot up. This might take a while, but when it's ready to use you should see something similar to 'Task (runnable)' as the screenshot below shows

You should now be able to access the app via a web browser of your choice (internet explorer, Google chrome, Firefox etc). Try opening a web browser and go to the webpage 'localhost:port' (where you replace 'port' with the integer that you specified for the *port* input variable)(please notice the ':'!!!). **The web application might take 1-2 minutes to load.** This is normal and this very slow loading speed will

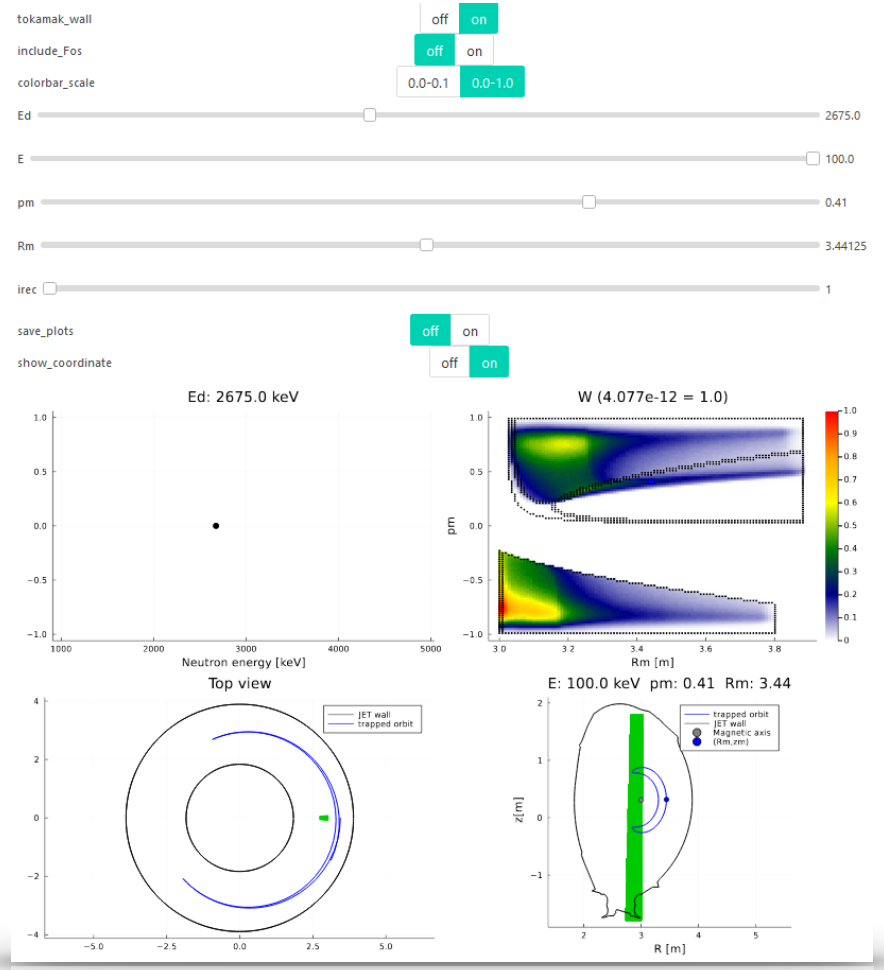


An example of how it might look when weightsWebApp.jl has finished loading and is ready to use

hopefully be improved in the future. Once it's finished, you should see something similar to the screenshot below.

There are several sliders and buttons which you can manipulate to view the orbit weight functions, the diagnostic, the tokamak, the data and the orbits. Let's go through them one by one.

- **tokamak wall** - [off/on] - This switch includes the tokamak wall in the poloidal cross-sectional plot (bottom right). It draws its data from the .eqdsk/.geqdsk file.
- **include_Fos** - [off/on] - If you have included a file path specification to the *filepath_Fos3D* input variable and set *plot_Fos* to true, you can choose to plot the orbit-space fast-ion distribution slice for the current fast-ion energy as



A normal view of the weightsWebApp.jl in a web browser

well. This can be useful if you want to visualise a particular orbit-space fast-ion distribution while at the same time visualising an orbit weight function, and their point-wise product WF.

- **colorbar scale** - [0.0-0.1/0.0-1.0] - You can quickly switch between two different colorscales by easily choosing one of the two options via the button switch. This can be useful

for identifying areas of faintly visible sensitivity in particular fast-ion energy slices.

- **phase space** - [(E,pm,Rm)/(E,mu,Pphi;sigma)] - If you have set *enable_COM* to true, you can use this button to switch between viewing slices of constant fast-ion energy in (pm,Rm) coordinates or in (Pphi, mu; sigma) coordinates.

Then you have the energy, pitch maximum and radius maximum sliders, which you can use to visualise different orbit weight function slices, as well as different orbits in relation to them. The (E, p_m, R_m) coordinate will be visualised as a dot in the slice plots.

- **save plots** - [off/on] - With this button, you can save all the four subplots in .png format. Just switch it from 'off' to 'on' and they will be saved in the OWCF folder. BEWARE! The app will save a set of four plots EVERY TIME YOU CHANGE ANY OF THE SLIDERS OR BUTTONS. So, when you have saved the plots you want, be sure to reset the *save plots* button to 'off'. Also, please note, that the topological boundaries will grow slightly when *save plots* is set to 'on'. This is to adjust for the difference in resolution between plotting and saving. Also, please move the saved .png files out of the OWCF folder afterwards, to keep the OWCF folder clean. Alright, so the subplots that you see in the figure on the previous page are the following:
- **Top-left:** A plot keeping track of the currently viewed diagnostic energy bin.
- **Top-right:** This is where the magic happens. This is the current orbit-space energy slice, including topological regions. It shows the orbit-space sensitivity for that particular slice. It also indicates the current phase-space coordinate with a sole coloured dot (whose colour depends on the currently highlighted topological region).
- **Bottom-left:** This subplot shows a view of the orbit path for the current phase-space coordinate from above the tokamak (with the HFS and LFS walls clearly visible).

- **Bottom-right:** Finally, this last subplot visualises a cross-sectional projection of the orbit path for the current phase-space coordinate. If provided, it also visualises the diagnostic sightline of the orbit weight functions. It also visualises the tokamak wall if specified with the *tokamak wall* switch.

If something goes wrong with the app, it's always best to re-start Julia by typing 'exit()', hit enter and then type 'julia' and hit enter, in the terminal/powershell, and re-load the app. (Hint: To get out of an error message, or shut down the app from the terminal, press ctrl + c)

There you have it.
Best of luck with using the weightsWebApp.jl!

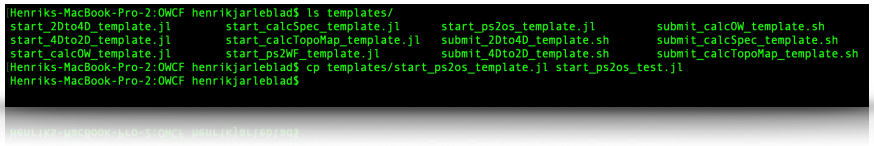
5.
TRANSFORMING FAST-ION DISTRIBUTIONS

- ps2os.jl

In addition to being able to compute orbit weight functions, this might be one of the most important features of the OWCF. Because it actually lets you take distributions from normal (E, p, R, Z) space, henceforth referred to as *particle space*, and transform them into (E, p_m, R_m) orbit space.

How do we do it? Well, as with almost everything in the OWCF, we do it via copying a start file from the OWCF/templates/ folder, specifying the input variables and then execute it. Let's see what we need to do.

First, copy the *start_ps2os_template.jl* into the OWCF/ folder as shown below. Either via the terminal/Powershell or via a Finder/File explorer window. As the figure shows



Copy the *start_ps2os_template.jl* file from the OWCF/templates/ folder to the OWCF/ folder, and rename it

Then, as earlier, use a code editing program of your choice to edit the start file. First, you will see the file description and system specification variables, just as before. Then, the actual input variables are

- **include1Dto3D** - If true, then the vectorised orbit-space distribution (in 1D) will be inflated to its true 3D form (in

addition to having been transformed to orbit space). The distribution that is the output of the ps2os.jl script will be in a 1D vector format, where every element corresponds to a valid orbit for the orbit grid you have specified.

- **filepath_distr** - The file path to the (E, p, R, Z) distribution that you want to transform into (E, p_m, R_m). It can be either in .h5/.hdf5 file format or in .jld2 file format. The only requirements are the names of the keys to the data fields. Please see the *ps2os.jl* file description for more info.
- **filepath_eqdsk** - We've covered this one before.
- **filepath_W** - It is possible to specify a .jld2 file with orbit weight functions (computed with *calcOrbWeights.jl*) from which the algorithm will try to extract information about the fast-ion species and (E, p_m, R_m) orbit grid (ranges and dimensions). This can be convenient if you want an (E, p, R, Z) distribution transformed onto the same (E, p_m, R_m) orbit grid that you have used to compute orbit weights for.
- **flip_F_ps_pitch** - If true, then the algorithm will mirror the (E, p, R, Z) distribution in pitch before transforming to orbit space. That is, it will flip the pitch sign of the distribution. $f(E, p, R, Z) \rightarrow f(E, -p, R, Z)$ and vice versa. Should most often be set to false.
- **folderpath_o** - We've covered this one before.
- **numOsamples** - The number of Monte-Carlo samples that you want to use to transform the (E, p, R, Z) distribution to orbit space. The algorithm samples from the particle-space distribution and the samples are binned onto the orbit-space grid that you have specified. Usually, I would recommend to use 10 times as many samples as you have valid orbits for your orbit-space grid. A good rule of thumb for how many valid orbits there are for a given orbit-space grid is to take the total number of grid points (for example $33 \times 31 \times 32 = 32736$), multiply it by a factor of 0.67 and then you have a good estimate for the number

of valid orbits for that particular orbit-space grid ($0.67 \times 32736 \approx 21900$).

- **nbatch** - This input variable will determine how often the algorithm saves the transformation progress. In the old days, before the performance upgrade, it used to take an hour or more to transform between reasonable (E, p, R, Z) and (E, p_m, R_m) grids. Then, to have the algorithm regularly save the progress in case it would be shut down, was a good idea. Nowadays however, it is basically superfluous. Please set it to 1/10 of the number of Monte-Carlo samples (*numOsamples),* or similar.
- **os_equidistant** - If true, the algorithm will assume that the orbit-space grid that you are transforming onto has equidistant grid points. False is currently not supported.
- **folderpath_OWCF** - We've talked about this one before.
- **timepoint** - The timepoint of the tokamak shot. Specified as a string with the format "XX,YYYY" where XX are seconds and YYYY are decimals. Please not the use of "," and not "." to separate the seconds and decimals.
- **tokamak** - Specify the name of the tokamak that you are simulating. It is merely for aesthetic purposes.
- **TRANSP_id** - Specify the TRANSP ID of the tokamak shot that you are simulating. In ps2os.jl, it is merely for aesthetic purposes (however, please note that that is not the case in other OWCF scripts, such as ps2WF.jl for example).
- **useWeightsFile** - We've covered this one before.
- **verbose** - You know what this one is about.
- **visualizeProgress** - This one as well.

Then we have some input variables that you only need to specify if **useWeightsFile** is set to false. They should be self-explanatory.

Then, you can also choose to have the fast-ion distribution interpolated in (E, p, R, Z) prior to transforming it into (E, p_m, R_m).

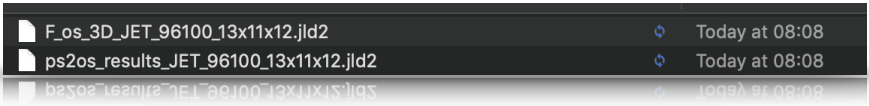
- **interp** - If true, the algorithm will interpolate the particle-space fast-ion distribution onto an (E, p, R, Z) grid with the same end points as the original one (in the .h5/.hdf5/.jld2 file) but with the dimensions nE_ps x np_ps x nR_ps x nZ_ps.

Alright, that should be all of them. Specify them as necessary and then start the script in the usual way

1. Starting the terminal/powershell
2. Navigate to the OWCF folder
3. Start Julia by typing 'julia' and hit enter
4. Type 'include("start_ps2os_test.jl")' and hit enter

Or whatever equivalent you have named your start file to (as stated in previous chapters, it doesn't have to be named 'test'. In fact, it shouldn't be, as I have written about earlier).

Then, if all goes well, after the script has finished you should have the transformed distribution in the *folderpath_o* output folder as you requested.



Output files from the ps2os.jl script

Please note, depending on the grid sizes and number of Monte-Carlo samples, the transformation might take a really long time. Usually, when I run it with an (E, p, R, Z) grid size of approximately $200 \times 50 \times 60 \times 60$ and an (E, p_m, R_m) grid size of approximately $70 \times 40 \times 50$, 2000000 Monte-Carlo samples and 40 computing cores on my university cluster, it takes roughly 15-20 minutes (including start-up and loading packages).

When I run it on my Macbook pro with 2,7GHz Intel quad-core i5 with an (E, p, R, Z) grid size of $100 \times 50 \times 43 \times 47$, 10000 Monte-Carlo samples, it takes roughly 3 minutes (including start-up and loading packages).

The two output files that you've got are the orbit-space distribution in 3D and 1D format, respectively. You can use the 3D quantity to specify the *filepath_Fos3D* input variable in weightsWebApp.jl, for example, for visualization. You can also use the distrWebApp.jl to visualise only the fast-ion distribution.

That concludes this chapter on how to transform from (E, p, R, Z) particle space to (E, p_m, R_m) orbit space.

6.
COMPUTING FORWARD MODEL SIGNALS

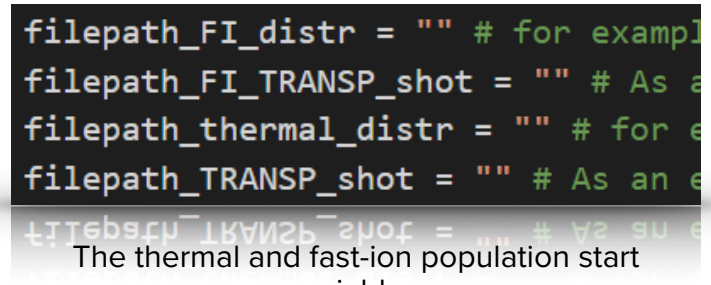
- calcSpec.jl

Another feature of the OWCF is the ability to compute forward model signals S. This can be useful if you would like to compare S with the WF and confirm that they match. For this feature, you use the calcSpec.jl script via the start_calcSpec_template.jl start file. By this point, you should be familiar with how to copy a start file from the OWCF/ templates/ folder to the OWCF/ folder and rename it appropriately. Do that, and then let's take a look at the input variables together. First, we see the system specification variables, as per usual. I will not do a walkthrough of the familiar variables (such as *diagnostic*). However, some of the input variables, previously not covered here, include:

- **Ed_min** - This is the lower energy bound of your diagnostic energy grid, and corresponds to $E_{1,d}$ in *H. Järleblad et al* (Nucl. Fusion, 2022)
- **Ed_max** - This is the upper energy bound of your diagnostic energy grid, and corresponds to $E_{n,d}$ of *H. Järleblad et al* (Nucl. Fusion, 2022)
- **Ed_diff** - This is a new kind of input variable! Instead of specifying the number of grid points (as you have done with fast-ion energy, pitch maximum etc) you now have to specify the size of all your diagnostic energy bins (in keV). Only a float or int is accepted, since the script assumes all diagnostic energy bins to be of the same size. Please note! This variable will then indirectly determine the number of

diagnostic energy grid points/the resolution of your diagnostic signal.

Then, we come to these rather weird pairs of input variables



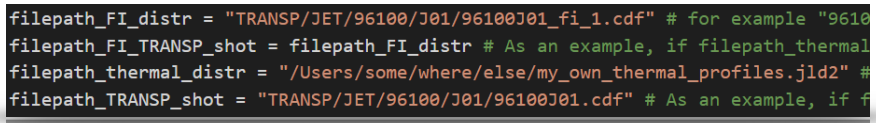
The thermal and fast-ion population start variables

You are right to be confused. However, as it happens, it is these input variables that will specify the fast-ion and thermal populations for your fusion reaction. You might be asking why we need four input variables then; should it not be enough with just two? You are absolutely correct. It should be! Unfortunately, the way the OWCF is constructed, as of this version (v1.0), it is prevented from working with just two input variables, in case you want to use TRANSP data for either the fast-ion or thermal distributions. Let's examine why this is the case.

First things first, if you do not want to use any TRANSP data for your forward model signal computation, you only have to specify *filepath_FI_distr* (which can then be either a .h5 file or a .jld2 file) and *filepath_thermal_distr* (which can then be either a .jld2 file or simply ""). If this is what you want, simply leave *filepath_FI_TRANSP_shot* and *filepath_TRANSP_shot* as "".

Now, let's say that you want to use a TRANSP file in .cdf file format for the fast-ion distribution, but a .jld2 file (or simply "") for the thermal distribution. Then, you still have to specify

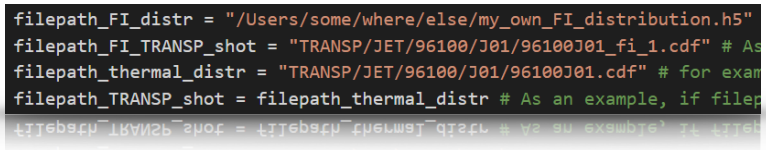
the TRANSP shot file pertaining to your fast-ion TRANSP file with the *filepath_TRANSP_shot* variable. It might look like this



Example of how to specify TRANSP files for calcSpec.jl

Notice how the files specified for the *filepath_FI_distr* and *filepath_TRANSP_shot* variables are from the same TRANSP run. Also notice how the *filepath_thermal_distr* variable is specified to be a .jld2 of your own, completely independent from the TRANSP data. Finally, notice how the *filepath_FI_TRANSP_shot* is specified to be the same as the *filepath_FI_distr* variable, since it is a TRANSP fast-ion file.

Let's continue. Let's say that you want to use TRANSP data for the thermal distribution, but a .h5 file (or .jld2 file) for the fast-ion distribution. Then, it might look like this



Notice how the files specified for the *filepath_thermal_distr* variable and *filepath_FI_TRANSP_shot* are from the same TRANSP run. **The reason that you need to specify a fast-ion TRANSP file even though you only want to use TRANSP thermal profiles is that the fast-ion TRANSP file contains information about which time window of the TRANSP shot that you want to access data for.**

Finally, let's look at when you might want to use TRANSP data both for the fast-ion and thermal distribution. It might look like this

```
filepath_FI_distr = "TRANSP/JET/96100/J01/96100J01_fi_1.cdf"
filepath_FI_TRANSP_shot = filepath_FI_distr # As an example,
filepath_thermal_distr = "TRANSP/JET/96100/J01/96100J01.cdf"
filepath_TRANSP_shot = filepath_thermal_distr # As an example
```

Another example of how to specify TRANSP files

And finally, for good measure, let's take a quick look at what it might look like when you don't want to use any TRANSP data at all

```
filepath_FI_distr = "/Users/some/where/else/my_own_FI_distribution.jld2" # for
filepath_FI_TRANSP_shot = "" # As an example, if filepath_thermal_distr=="96100
filepath_thermal_distr = "/Users/some/where/else/my_own_thermal_profiles.jld2"
filepath_TRANSP_shot = "" # As an example, if filepath_FI_distr=="96100J01_fi_1
```

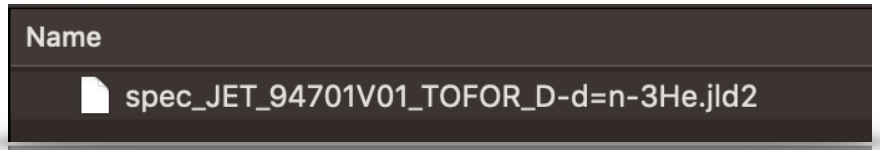
No TRANSP data used (and needed)

Alright, I hope you have got a feeling for how to specify the input variables in case you want to use TRANSP data. Now, let's move on to some other input variables.

- **mc_samples** - The number of Monte-Carlo samples to be used for sampling from the fast-ion distribution. Usually, around a million should be enough for a smooth signal.
- **mc_chunk** - The size of the chunks in which to divide the Monte-Carlo sampling. Usually, 10000 is a good number. If you have a very large amount of RAM (per core) then you may crank it up. This input variable exists to make the computations as efficient as possible while still not using too much RAM. The bigger the amount of RAM you have, the closer to *mc_samples* you may specify the value of this variable to be.
- **noiseLevel_b** - Background noise level for your signal.
- **noiseLevel_s** - Signal noise for your signal.

- **calcProjVel** - If true, then the OWCF will compute a spectrum of projected velocities for the given magnetic equilibrium and diagnostic viewing cone.
- **thermal_temp_axis** - If you have specified the *filepath_thermal_distr* input variable to "", then a default temperature profile will be used (please see OWCF/misc/temp_n_dens.jl). You have the option to specify the temperature on axis for this temperature profile. This you do with this input variable.
- **thermal_dens_axis** - Same as for the *thermal_temp_axis*, but with the thermal density profile instead.

As with the ps2os.jl script, you can choose to have the fast-ion distribution interpolated in (E, p, R, Z) before sampling from it to transform it to (E, p_m, R_m). So go ahead, please specify all the input variables in the start file, and execute the start file in the normal way (start Julia, type 'include("start_calcSpec_test")' and hit enter). When it is done, you should have a file in the *folderpath_o* output folder similar to what the screenshot below shows.



Neutron spectrum signal computed for JET tokamak equilibrium, TRANSP ID 94701V01, for the TOFOR diagnostic and fusion reaction $D(d,n)^3He$

7.
COMPUTING WF SIGNALS

Ok, this is going to be easier than you might think. All you need to do is to load a weights file and a distribution file and multiply the weight matrix with the distribution vector. Let's take a look at how it works.

First, start by loading the weight matrix from the .jld2 file that you got as output from calcOrbWeights.jl (chapter 2). As the screenshot below shows.

```
using JLD2
myfile = jldopen("orbWeights_JET_96100J01_D-d=n-3He_13x11x12.jld2",false,false,false,IOStream)
W = myfile["W"]
Ed_array = myfile["Ed_array"]
close(myfile)
```

How to load a weight matrix and the diagnostic energy bins. JET and TOFOR diagnostic used as example

The .jld2 file also contains other data but we only need the *W* and *Ed_array* for this example. Next, you are gonna need the data from the .jld2 file that you got as output from ps2os.jl (chapter 5). As the screenshot below shows (if you have typed 'using JLD2' (or any other Julia package for that matter) once during a Julia session, the package is loaded and you do not need to type it again. Just like Python!).

```
myfile = jldopen("ps2os_results_JET_96100_13x11x12.jld2",false,false,false,IOStream)
F_os_raw = myfile["F_os_raw"]
nfast = myfile["nfast"]
F_os = (nfast/sum(F_os_raw)) .*F_os_raw
close(myfile)
```

How to load a vectorised orbit-space distribution and correctly normalise it.

When the vectorised orbit-space distribution was saved to file by the ps2os.jl script, it was saved as *F_os_raw*. It simply means that all the binned samples were saved as integers in their respective bins, sum(F_os_raw)=*number of valid samples*. So, to get our actual vectorised orbit-space distribution *F_os*, we have to multiply by the total number of fast ions of our distribution *nfast*, and divide by the sum of the vector *F_os_raw*.

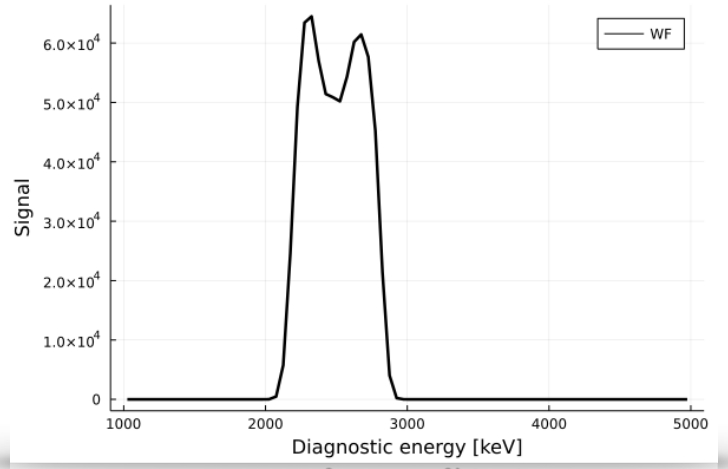
Now that you have both the weight matrix loaded as *Wtot* and the fast-ion orbit-space distribution loaded as *F_os*, go ahead and multiply them together to get your diagnostic signal *S_WF* as the screenshot below shows.

```
S_WF = Wtot * F_os
```

The famous $S = WF$

You can then plot your diagnostic WF signal as the screenshot at the top of the next page shows. If you are wondering how to set the labels for the x- and y-axes, as well as other plot-related quantities, please consult the official page for the Julia Plots.jl package (which can be found here https://docs.juliaplots.org/latest/). It should be mentioned however, that you do not need to use the Plots.jl package. You could also use other plot packages, such as the PyPlot.jl package.

Anyhow, that concludes this chapter on how to compute and plot WF signals the normal way. But what if you run out of RAM memory because you need really high-resolution weight functions? Don't worry! The next chapter is for you.



Example plot of a WF signal

8.

COMPUTING ULTRA HIGH-RES WF SIGNALS AND THEIR ORBIT FRACTIONS

- ps2WF.jl

We are now going to talk about one of the most advanced scripts in the OWCF. Let's say that you have a fast-ion distribution with an ion-cyclotron resonance heating (ICRF) energy tail that goes up into the several MeVs. Let's say it goes up to 2500 keV. You also have a diagnostic which needs to be quite high-resolved in terms of orbit-space grid points. Let's say you need an energy resolution no coarser than a couple of keVs. And, to make matters even more interesting, you also need to have a p_m and R_m grid resolution of at least 100×100. That would mean that you need an orbit-space grid with an approximate total of $2500 \times 100 \times 100 = 25000000$ (E, p_m, R_m) points. Let's say the orbit validity ratio of our orbit space is 0.67. That would mean $25000000 \times 0.67 = 16750000$ valid orbits. And for every valid orbit, you would like to compute an expected diagnostic signal with 80 diagnostic energy bins. That would mean that your weight matrix would contain $80 \times 16750000 = 1340000000$ matrix elements. And every floating point number is stored as a 64 bit. Then your orbit weight matrix would require $1340000000 \times 64 = 85760000000$ bits of memory storage. That's over 10 GB of required storage... Yikes! Surely, there must be a better way?

There is! Introducing: ps2WF.jl. With this script, you specify a fast-ion (E, p, R, Z) distribution and an ultra-fine orbit-space grid. The algorithm will then split the WF signal computation into many smaller parts. By doing so, **it can compute diagnostic signals using orbit weight functions with a virtually infinite orbit-space grid resolution**. How does it do this, and why does it work? It all comes down to the following equation

$$WF = W_1F_1 + W_2F_2 + \ldots + W_nF_n\,.$$

Here, the WF computation has been split into many smaller parts. This is what the ps2WF.jl algorithm does, specifically in fast-ion energy (it's usually the dimension that needs an ultra-high resolution grid).

Before we take a look at all the input variables, let's talk about one more awesome feature of the ps2WF.jl script. **It can compute orbit type fractions!** What does that mean? Basically, it means that the ps2WF.j script can split the fast-ion distribution, the orbit weight functions and the diagnostic signal into their respective orbit type components. This is possible because (using fast-ion energy as an example) we can always split quantities into their orbit type constituents

$$\begin{aligned} f(E) = & f_{\text{co-passing}}(E) + f_{\text{counter-passing}}(E) \\ & + f_{\text{trapped}}(E) + f_{\text{potato}}(E) \\ & + f_{\text{stagnation}}(E) + f_{\text{counter-stagnation}}(E), \end{aligned}$$

$$\begin{aligned} w(E) = & w_{\text{co-passing}}(E) + w_{\text{counter-passing}}(E) \\ & + w_{\text{trapped}}(E) + w_{\text{potato}}(E) \\ & + w_{\text{stagnation}}(E) + w_{\text{counter-stagnation}}(E) \end{aligned}$$

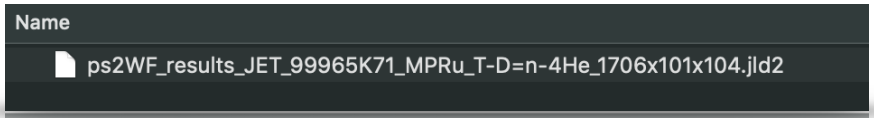
and similarly for p_m and R_m. We can also split the WF-signal (which is a function of diagnostic energy E_d) into its orbit type constituents:

$$WF = (WF)_{\text{co-passing}} + (WF)_{\text{counter-passing}} + (WF)_{\text{trapped}} + (WF)_{\text{potato}} + (WF)_{\text{stagnation}} + (WF)_{\text{counter-stagnation}}$$

These constituents can then be visualised using e.g. signalWebApp.jl as we shall see in later chapters. Now, let's take a look at some of the ps2WF.jl input variables (I will not go through input variables that we've covered in previous chapters. Please see previous chapters for a description of those).

- **calcWFOs** - If true, the ps2WF.jl will compute orbit type fractions as well and save them in the results file. Please note that you need to set *inclPrideRockOrbs* to true for the results file to be readable by *signalWebApp.jl.* Also, please note, to compute the orbit type fractions takes extra computational resources. So if you don't care about the orbit type fractions and want to minimise computational resource usage, set this input variable to false.
- **nEbatch** - This input variables sets the size of the fast-ion energy grid partitioning. For example, if set to 2, every chunk in the equation on the previous page will be of size $2 \times npm \times nRm$. The more RAM you have, the larger this chunk can be.
- **og_filepath** - If you, for some reason, pre-computed a huge (!) orbit grid that you want to use for the ps2WF.jl script, you can use it by specifying this input variable. It should be a string with the path to a calcOrbGrid.jl output file.

So go ahead and specify all the input variables. If you have done it correctly, the script should start and you might want to go to sleep for this one. It's going to take quite some time (even with multi-core processing)! When it's done, you should see the following file in your *filepath_o* output folder, as the screenshot below shows.



Example of an output file from ps2WF.jl

Right, there is also one more thing you need to know about the ps2WF.jl. The script will create a progress file every time it has completed an *nEbatch*. This is to ensure it doesn't have to start the whole computation from the beginning if it is unexpectedly interrupted. This progress file will be deleted when the computation is finished and the results files created.

That should be it. Good luck using the ps2WF.jl script!

9.

USING THE OWCF ON SUPERCOMPUTERS

- The templates/submit_....sh files

This is going to be fun! So far, I have only told you how to use the OWCF on local computers (a single machine, basically). But, what if you wanted to use the OWCF on a computational cluster, such as a supercomputer? The DTU Physics supercomputer Niflheim is an example of such a cluster. If that is the case, then we need to add some more steps before we execute our scripts. The basic outline is as follows

- You still specify the input variables in a start file
- You have to specify the computational cluster batch job information in a submit (.sh) file
- You have to make sure that the number of cores that you specify in your submit file matches the number of cores that you specify in your start file

All default submit files in the OWCF can be found in the OWCF/templates/ folder. They are customised to be used on computational clusters using the SLURM Workload Manager system (Niflheim uses SLURM). If you would like to use the OWCF on computational clusters with other workload managers (such as PBS or PBS Pro) then you need to write your own submit (.sh) files. Alright, let's take a look at a submit file. Let's use the *submit_calcOW_template.sh* submit file as an example (all submit files are similar anyway). So go head an open that file in a code editing program of your choice.

```
#!/bin/bash
#SBATCH --mail-type=ALL
#SBATCH --mail-user=henrikj@dtu.dk
#SBATCH --export=ALL
#SBATCH -J calcOW
#SBATCH --partition=xeon40
#SBATCH -n 40
#SBATCH -N 1
#SBATCH --mem=300G
#SBATCH --time=2-00:00:00
#SBATCH --output=log_file_calcOW_template.out

julia start_calcOW_template.jl
mv start_calcOW_template.jl ../OWCF_results/template/
mv log_file_calcOW_template.out ../OWCF_results/template/
mv submit_calcOW_template.sh ../OWCF_results/template/
```

An example of a submit file

The first thing you will see will be lines with '#' in front of them. These are lines that will be sent as specification requests to the workload manager. Basically, you specify how you want the computational cluster to run your job. Then, you will find lines further down that do not have the # symbol in front of them. These are lines that will actually be executed by your computational cluster job. Alright, let's go through the lines in the submit file, one by one.

- **#!/bin/bash** - The first line means that you are declaring this file to be executed via the bash command. There are other command line protocols/languages to execute commands with from the Linux command line. The bash command is one of them (and the most common?).
- **#SBATCH —mail-type=ALL** - This line tells the SLURM workload manager for which type of computational job events you would like to receive emails. 'ALL' simply means you will receive emails whenever the job starts, completes, if it is cancelled and/or if it fails.

- **#SBATCH —mail-user=henrikj@dtu.dk** - This line tells the workload manager to what email address it should send the emails that will notify you of job events.
- **#SBATCH —export=ALL** - This line tells the workload manager that you want to export all your .bashrc (or .bash_profile) variables and features to all external processes (all computer cores that will perform your multi-core computations). The ability to simply write 'julia' and have Julia start is one those features. So this line is extra important.
- **#SBATCH -J calcOW** - This line tells the workload manager what you want to name your computational job. It doesn't have to be 'calcOW' then of course. It can be anything! Let your imagination run free!
- **#SBATCH —partition=xeon40** - This line tells the workload manager that we would like to use the 'xeon40' computational cluster partition. You see, all (I think?) supercomputers are divided into partitions. Partitions are basically like small clusters within the super-computer cluster. The partition can have specialised features and abilities (such as number of cores per machine, amount of available RAM etc). Unless you want to use the default partition, you have to tell the workload manager which partition you want to use. On the xeon40 Niflheim partition, every computer/machine has 40 cores each. So, that brings us to our next line...
- **#SBATCH -n 40** - ...which tells the workload manager how many cores you would like to use for your computational job. Notice how xeon40 has '40' in its name, and I have set the '-n' flag to precisely 40? This is because this line (containing the '-n' flag) must (well...) be a multiple of the number of cores available on each machine on the partition. Basically, N × *number of cores on a single machine* = n. How do we specify 'N' you might ask? The next line...

- **#SBATCH -N 1** - ...takes care of that for you. So if instead I had specified the '-N' flag to be '2', then I would need to have the '-n' flag to be '80'.
- **#SBATCH —mem=300G** - This line tells the workload manager how much RAM memory you would like to use per computer (machine or node). So if you have specified 'N' to be 2, then you will get a total of $N \times 300 = 2 \times 300 = 600$ GB of RAM ('G' means gigabytes). Getting the right amount of RAM required for certain computational jobs is really an art form in itself. But make sure that you read up on what is the maximum amount of RAM memory per computer (can also be called machine or node) that your computational cluster partition has available, so that you do not request more RAM memory than is available.
- **#SBATCH —time=2-00:00:00** - This line tells the workload manager the time limit of your computational job. If your job has not completed by then, it will be automatically terminated. The time format is [DAYS]-[HOURS]:[MINUTES]:[SECONDS].
- **#SBATCH —output=log_file_calcOW_template.out** - This line specifies the name of the output file that your computational job will produce. All program print and terminal output will be written to this file. It can be named whatever you want, and does not even have to have the file extensions of '.out'. It can be named 'myAwesomeCupcake.what' for example and no errors would be thrown.

Alright, now we have talked about the '#SBATCH' lines. Please note, that '#SBATCH' must always be written at the start of those lines for the workload manager to register them. Now let's talk about the other lines below.

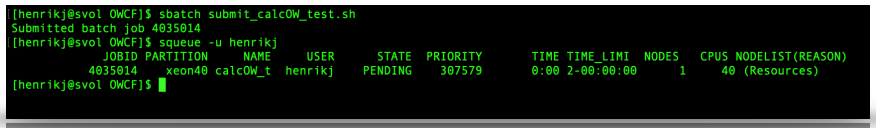
All the lines that do not have '#SBATCH' in front of them should be written below all '#SBATCH' lines. They will be lines that the computational job will actually execute. So for example 'julia start_calcOW_template.jl' will have Julia execute the *start_calcOW_template.jl* file, just as per usual.

You simply have to do it via a submit file now! The lines starting with ‘mv’ tells the computational job to move the specified files to the folder specified after. So the syntax is ‘mv [FILE] [OUTPUT FOLDER]’.

When you run the OWCF with a submit file, please change all instances of ‘template’ to whatever you prefer to identify your computational job with. That includes the output folder to which the ‘mv’ move command will move the start-, log- and submit-files.

Alright, so go ahead and specify all the lines in the submit file. When you are done, save it. Make sure that you have also specified everything you need in your start file. Please note, every script (and start file) that we have so far talked about in the OWCF has a corresponding submit file. Make sure that you copy both the start file and the submit file from the OWCF/templates/ folder into the OWCF/ folder. Maybe obviously, it does not have to be the calcOW script. That is only used as an example. Please take a look in the OWCF/templates/ folder to see which submit files are available.

Once you have copied both the start and submit files into the OWCF/ folder as well as specified and saved both of them, you are ready to start the computational cluster job. Make sure that you are standing in the OWCF/ folder, then type ‘sbatch submit_calcOW_template.sh’ (or whatever other submit file you have specified) and hit enter. *Please note! The ‘sbatch’ command only works on SLURM Workload Manager computational clusters.* Your job will now either have started right away, or it will be in queue and start computing shortly. As the screenshot below shows.



Submitted computational job in queue on a SLURM Workload manager computational cluster

The results files will then appear in the output folder that you specified with the *filepath_o* input variable in the start file, as per usual. The start-, log- and submit-files will be moved to whatever folder you specified last on the 'mv' lines in the submit file.

The computational jobs will also have generated one Julia log file for every core that you used for your job. If you an error occurs on a particular core for some reason, you can check the Julia log file for that specific core. The Julia log files will be numbered from '0' to 'n'. They will be created in the OWCF/ folder. If no error occurred during your computational job, go ahead and delete the Julia log files when your job has finished. Also, delete the Julia log files in general, as soon as you are done with them.

That should be it. Good luck with using the OWCF on computational clusters/supercomputers!

10.
EXTRACT NULL ORBITS

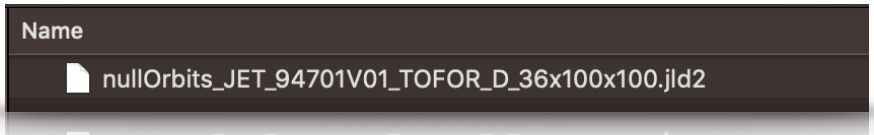
- helper/extractNullOrbits.jl

This script will identify null orbits given a diagnostic signal and a weight matrix, and save the null orbits in a results file. The default is to save the null orbits in a 2D matrix format, where each row corresponds to a specific weight function. So it's a composition of 1D arrays stacked row by row, basically. You could also set the input variable *include2Dto4D* to true. The algorithm will then convert all null orbits weight functions into their 3D inflated form, and since there is one weight function for every diagnostic energy bin, the resulting quantity will be 4D. Alright, let's have a look at some of the input variables:

- **signalNullThreshold** - This input variable is a float between 0.0 and 1.0. It will set the threshold for the algorithm to count a signal measurement as eligible for null orbits. If set to 0.01 for example, then every diagnostic energy bin with a signal amplitude equal to or below 0.01*maximum(signal) will be counted as diagnostic energy bins whose weight functions have null orbits.
- **orbNullThreshold** - This input variable is a float between 0.0 and 1.0. It will set the threshold for the algorithm to count a weight in the weight matrix, with a corresponding orbit, as a null orbit. If set to 0.01 for example, then every weight in the weight matrix with a value equal to or above 0.01*maximum(W) will be considered a possible null orbit.

- **filepath_W** - The filepath to the 2D output from the calcOrbWeights.jl script. Please note, the start file is requesting the 2D file, and not the 4D one.
- **filepath_S** - The filepath to the diagnostic signal. Could be an output of either the calcSpec.jl or ps2WF.jl script. The important thing is that the number of diagnostic energy bins of the signal (length(Ed_array)) corresponds to the number of rows of the weight matrix loaded from filepath_W.
- **FI_species** - You need to manually specify the fast-ion species. “D”, “T” etc.
- **include2Dto4D & filepath_eqdsk** - If you want the algorithm to save the 3D (4D) version of the null orbits, you have to set the *include2Dto4D* input variable to true, and specify the magnetic equilibrium with the *filepath_eqdsk* input variable. This is because, to transform from 1D to 3D, we need the orbits of the orbit-grid. And to compute the orbits of the orbit-grid we need the magnetic equilibrium.

Once you have copied the start file from the OWCF/templates/ folder, specified the input variables and executed the script via ‘include(“start_extrNullOrbs_test.jl”)’ (or equivalent) you should (within not too long) have the output file in your *folderpath_o* folder. As the screenshot below shows. But wait, “how do I visualise the null orbits?” you might be asking. Well, to do that you need to have run extractNullOrbits.jl with the input variable *include2Dto4D* set to true. You can then visualise the null orbits with the distrWebApp.jl or the weightsWebApp.jl, the web app that we shall now revisit!



An example of the output from the extractNullOrbits.jl script

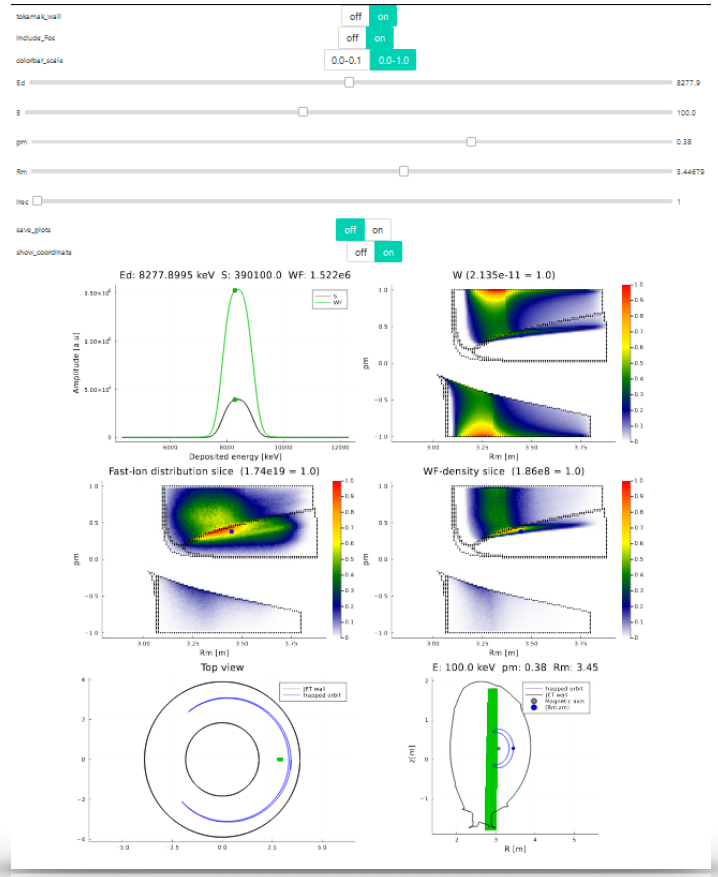
11.

VISUALISING ORBIT WEIGHT FUNCTIONS

- apps/weightsWebApp.jl (revisited)

I hope that you are sitting comfortably! We are now going to take a look at all the features of the weightsWebApp.jl, not just the visualisation of the orbit weight functions that we talked about in chapter 4. We are going to take a look at the full app. The weightsWebApp.jl can do a lot of other things, in addition to just visualising orbit weight functions. Let's look at some of the input variables for the weightsWebApp.jl once more:

- **filepath_tb** - The filepath to the topological boundaries of the orbit-space you want to visualise. Has to be an output of the calcTopoMap.jl script with *includeExtractTopoBounds* input variable set to true, or an output of the misc/extractTopoBounds.jl script.
- **filepath_W** - The filepath to the file containing the orbit weights to be visualised. Please note! It has to be the file containing the orbit weights in 4D format. Basically the output from calcOrbWeights.jl with *include2Dto4D* set to true or the output from orbWeights_2Dto4D.jl
- **plot_Fos & filepath_Fos3D** - If *plot_Fos* is set to true, you have to specify the path to an orbit-space fast-ion distribution in inflated 3D format via the *filepath_Fos3D* input variable as well. If you do, you will be able to visualise energy slices of the orbit-space fast-ion distribution along with the energy slices of the orbit weight functions.



A normal view of the complete weightsWebApp.jl

- **plot_S & filepath_spec** - If *plot_S* is set to true, you have to specify the path to a signal file via the *filepath_spec* input variable as well. If you do, you will be able to visualise where the diagnostic energy bin of interest is on the diagnostic signal as you examine the different orbit weight functions in your *filepath_W* data. Please note! The number of diagnostic energy bins used for your diagnostic signal stored in *filepath_spec* must

match the number of diagnostic energy bins/the number of orbit weight functions in your *filepath_W* file.

- **specFileJLD2** - If set to true, the app will assume that the signal file that you have specified with the *filepath_spec* input variable is in .jld2 file format. Otherwise, .h5 file format is assumed.
- **showNullOrbs & filepath_no** - If *showNullOrbs* is set to true, you have to specify the path to a file containing the 4D indices of the null orbits that you want to visualise. This file is specified via the *filepath_no* input variable. It can for example be computed via the extractNullOrbits.jl script and setting the *include2Dto4D* input variable to true
- **plot_S_WF & filepath_WF** - If *plot_S_WF* is set to true, you have to specify the path to a file containing the weight-computed synthetic signal WF via the *filepath_WF* input variable. This file can for example be the output file of the ps2WF.jl script.

If you specify all of the input variables above (and more), you can go ahead and start the app in the usual way. Once you do, and the app has finished loading, you should find a web app at the 'localhost:*port*' webpage that looks something like the screenshot above shows (after it has finished loading). Alright, let's go through the four plot windows:

- **Top-left:** Diagnostic (signal) energy plot. This subplot shows the currently selected diagnostic energy. As we shall see in later chapters, if you also have specified a signal S or a weight signal WF, it will also be visualised here, and the current diagnostic energy E_d will be highlighted.
- **Top-right:** This is where the magic happens. This is the current orbit-space energy slice, including topological regions. It shows the orbit-space sensitivity for that particular slice. It also indicates the current (E, p_m, R_m) coordinate with a sole coloured dot (whose colour depends on the currently highlighted topological region).

- **Middle-left:** This shows the fast-ion distribution for the current energy slice.
- **Middle-right:** This shows the WF density (the point-wise product of W and F) for the current energy slice. Regions of high values correspond to it being very likely that the diagnostic measurement (for the current diagnostic measurement bin E_d) originates from there.

There is one more thing that we need to talk about. If you set the input variable *showNullOrbs* to true and specified the *filepath_no* input variable, then the app will also show you the null orbits whenever the signal is zero (or close to zero). It will put a cross ('×') over all the weights that correspond to the null orbits for a specific diagnostic energy bin and fast-ion energy. That is it. That is all you need to know about the full weightsWebApp.jl. Have fun!

12.
VISUALISING ORBIT-SPACE TOPOLOGIES

- apps/orbitsWebApp.jl

To visualise orbit-space topologies and poloidal/toroidal transit times without orbit weight functions, fast-ion distributions etc., the user can utilise the **orbitsWebApp.jl**. The input variables includes the following

- **filepath_equil** - The path to the either an .eqdsk file containing the TRANSP magnetic equilibrium, or a .jld2 file computed with the compSolovev.jl helper script (located in the OWCF extra/ folder).
- **filepath_tm** - The path to the .jld2 file containing the topological map you want to visualise.
- **particle_species** - The particle species of the ion orbits you want to visualise. For example 'D', 'p', 'T', '3he' etc.
- **verbose** - Set to true if you want the app to talk a lot!
- **port** - The internet port where you will access your app.

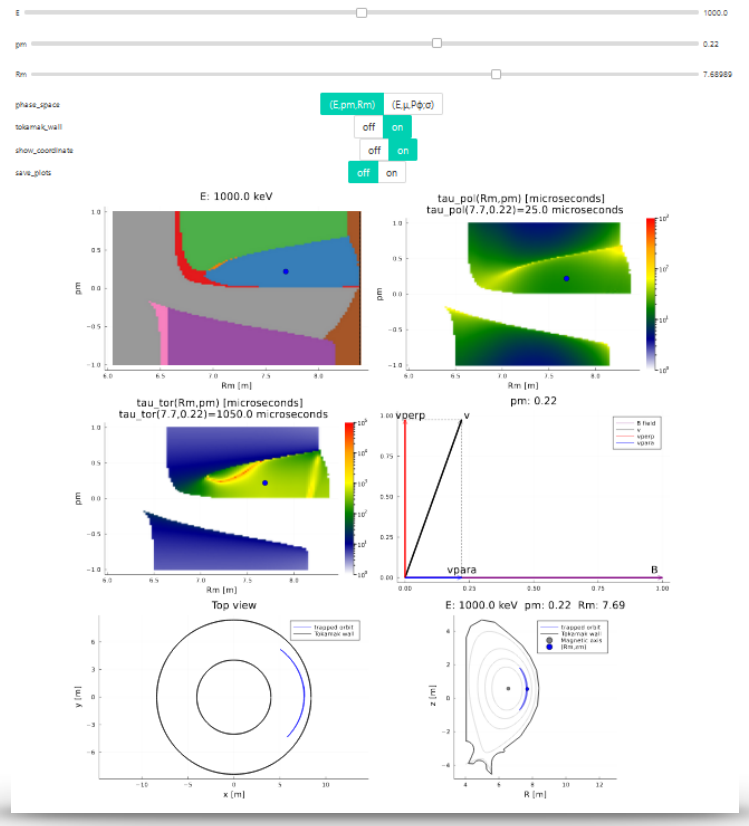
Don't forget to specify the important **folderpath_OWCF** variable above the regular input variables.

Once you have specified all the input variables, go ahead and start the app in the usual way: navigate to the OWCF folder path using the Windows Powershell or the Mac Terminal, start Julia by typing 'julia', hit enter', type 'include("apps/orbitsWebApp.jl")', hit enter, wait approximately 1 minute, go to the website 'localhost:[port]' in your web browser, wait for the app to load during approximately 1-2 minutes and voilà!. Replace '[port'] with the integer value of your *port* input variable. You should see an app similar to the

screenshot below. The app controls are similar to the app controls of the weightsWebApp.jl app. Remember, if you want to save plots, switch the *save_plots* control to 'on', wait 2-3 seconds and then switch it back to 'off'.

One more thing: in one of the plots of the orbitsWebApp.jl is a plot showing the parallel and perpendicular velocity vectors for the current (E, p_m, R_m) coordinate of interest. It's the p_m pitch that is visualised (the pitch at (R_m, Z_m)). The magnetic field line is shown in purple and the ion total velocity vector is shown in black.

Good luck, have fun with orbitsWebApp.jl!



A screenshot of the orbitsWebApp.jl

13.
VISUALISING SINGLE ORBITS IN DETAIL

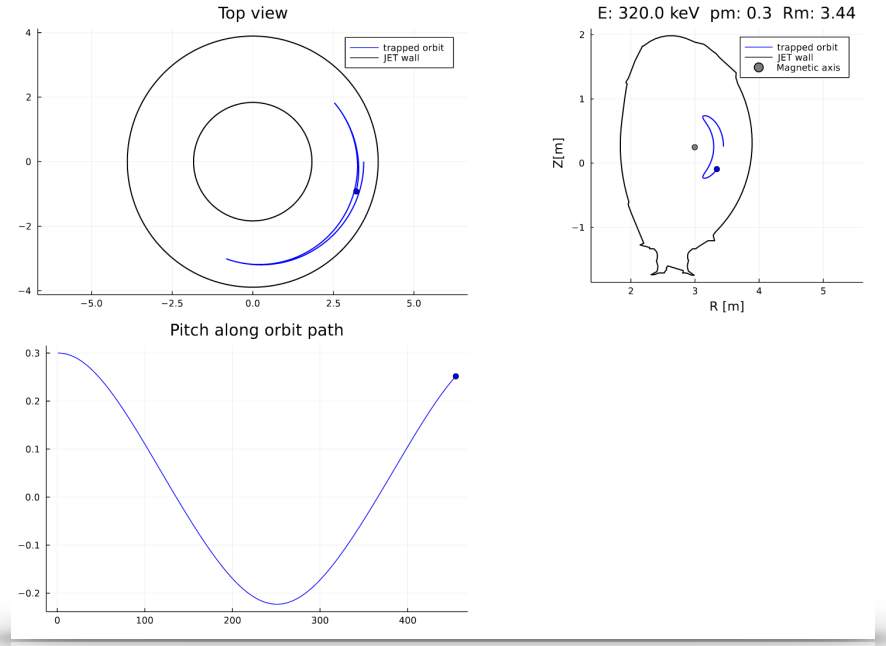
- apps/orbitWebApp.jl

Let's say that you would like to visualise a specific orbit in detail. For example, maybe you would like to examine how its pitch varies as the ion goes around the orbit? Wait no longer, because this is the app just for you! The input variables are really simple, we have covered them all in previous chapters. So go ahead and specify them, and boot up the app as per usual. Open a web browser of your choice, and go to the 'localhost:*port*' webpage. Wait 0,5-1 minute et voilà! You should see something similar to the screenshot below.

Please ignore all the warning messages you get in the Windows Powershell/Mac Terminal (it is just the orbit algorithm complaining).

In the app, you have familiar app controls such as *tokamak_wall, E, pm, Rm* and *save_plots*. However, *E, pm* and Rm are now given as boxes in which you can input any value (*E* in keV and Rm in meters). But there is also a new slider labelled *i*. What is this slider for? Well, it is with this slider that you choose the end point of the orbit. '0' is the starting point of the orbit ($t = 0$) and '500' is the point when the ion has completed a full orbit poloidally ($t = \tau_p$). In this way, you can examine how the ion moves along an orbit in detail. In addition, in the lower left-hand plot you will find a plot of the pitch of the ion as it moves along the orbit. This is useful because you can examine at what (R, Z) point the ion will have a specific pitch value, and also see how the pitch evolution looks like for various orbit types. For example, try

investigating how the pitch evolution looks like for all the different orbit types (co-passing, trapped, stagnation, potato, counter-passing and counter-stagnation).



Example screenshot of the orbitWebApp.jl

14.

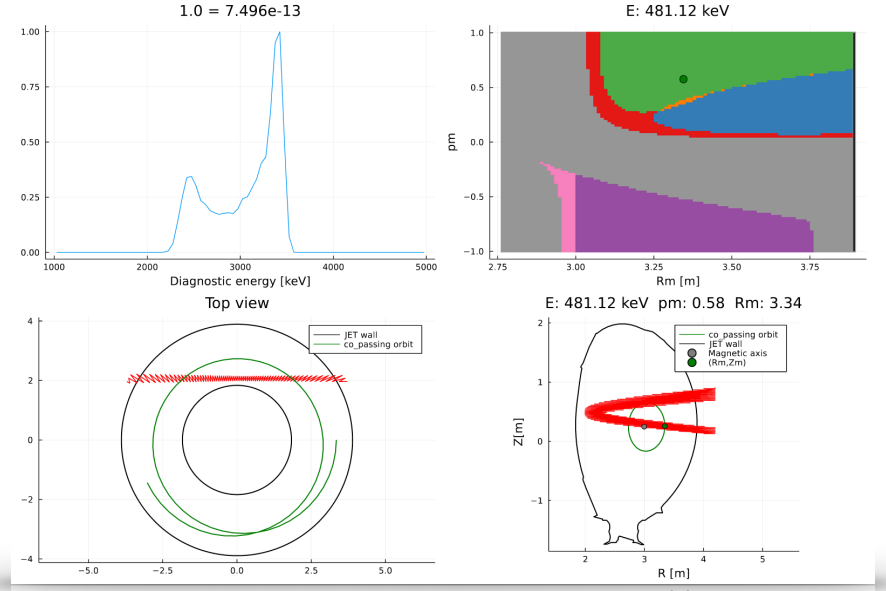
VISUALISING EXPECTED SIGNALS OF INDIVIDUAL ORBITS

- apps/weightWebApp.jl

Now this is interesting. We usually think of the weight matrix as a stack of rows of weight functions, where every weight function is a function of the three orbit-space coordinates as $w(E, p_m, R_m)$. **But what if we wanted to investigate what diagnostic signal a single orbit could be expected to produce?** (given a bulk plasma distribution and magnetic equilibrium of course). Well, that's what this app is for. Also, the input variables are nothing that you haven't seen before in this manual! So go ahead and specify them, and start the app in the usual way. When you access the app via the webpage 'localhost:*port*', you should see a screenshot similar to the one on the next page.

The (R_m, p_m) coordinate is shown in the upper right-hand plot, the topological map plot. The poloidal projections of the orbit, the tokamak wall and the diagnostic sightline is shown in the bottom right-hand plot. A top view plot of the orbit, the tokamak walls and the diagnostic sightline is shown in the lower left-hand plot. And finally, the most important, the expected diagnostic signal for the orbit with the (E, p_m, R_m) coordinate of interest is shown in the upper left-hand plot.

That is it for the weightWebApp.jl. Good luck, have fun!



Example screenshot of the weightWebApp.jl

15.
VISUALISING ORBIT-SPACE DISTRIBUTIONS ONLY

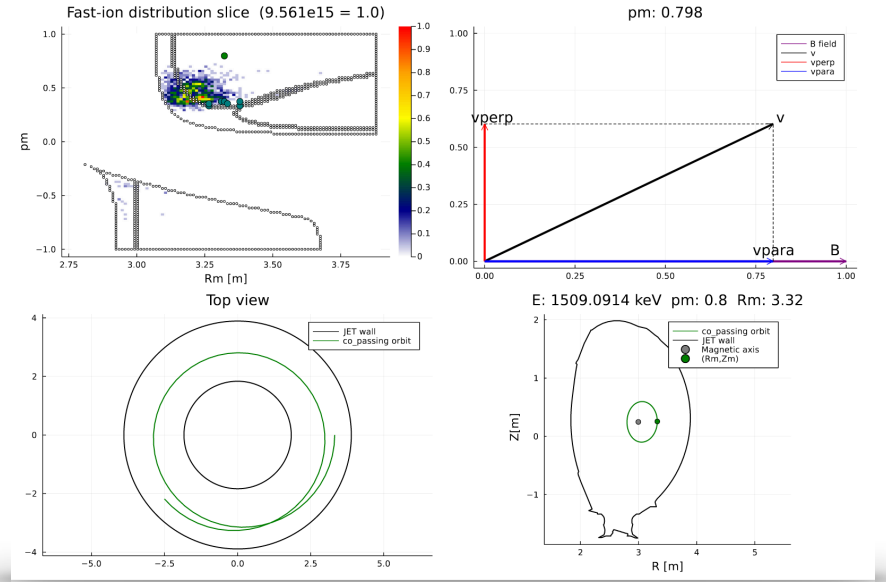
- apps/distrWebApp.jl

Let's say that you have only a three-dimensional orbit-space quantity. You don't have to worry about diagnostic energy bins or other things. You just want to examine this orbit-space quantity that depends on (E, p_m, R_m) and you want to examine it slice-by-slice in terms of fast-ion energy. How would you go about doing it? Introducing: The distrWebApp.jl! The input variables include

- **port** - Your usual port input variable that determines the web address on which the web app is hosted (webpage accessed via 'localhost:*port*' as usual).
- **verbose** - If true, then the web app will talk a lot.
- **filepath_tb** - The file path to the orbit-space topological boundaries. Can be an output .jld2 file from the misc/extractTopoBounds.jl script, or from the calcTopoMap.jl script with the input variable *includeExtractTopoBounds* set to true.
- **filepath_eqdsk** - The file path to the .eqdsk file containing tokamak magnetic equilibrium and boundary (wall).
- **filepath_distr** - The file path to the 3D orbit-space quantity that you want to visualize. Please note that it has to be a .jld2 file and contain the keys 'F_os_3D', 'E_array', 'pm_array' and 'Rm_array' where 'F_os_3D' is the 3D orbit-space quantity.

- **filepath_distr_2** - The file path to a second 3D orbit-space quantity that you would like to compare to the first. Apart from the same requirements as for the first file, this one has to have the exact same orbit-space grid as the first file too.
- **show_No & filename_no** - If *show_No* is set to true, and *filename_no* is specified to be a .jld2 file that is the output file of the extractNullOrbits.jl script, then the app will also plot null orbits, by super-imposing them on top of the slice plot of the 3D orbit-space quantity

So, go ahead and specify all the input variables, start the app in the usual way (see chapter 4 for reminder) and then when you connect to the 'localhost:*port*' webpage, you should see something similar to the screenshot below.



Example screenshot of the distrWebApp.jl

16.

VISUALISING ORBITS IN CUSTOMISABLE SOLOV'EV EQUILIBRIA WITH COM-COORDINATES

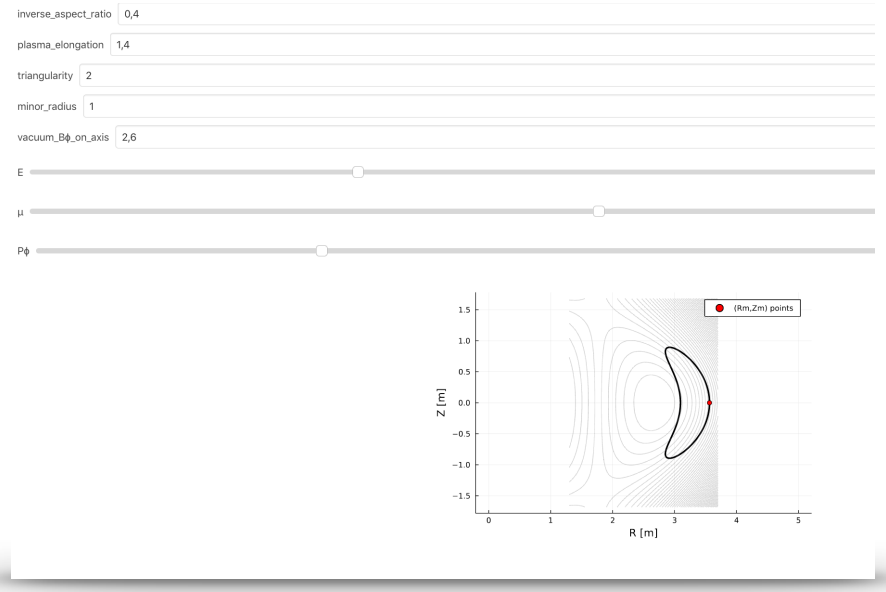
- apps/comWebApp.jl

Out of all the apps in the OWCF, this is the one that slightly sticks out. It's not very refined, but it can still be useful for certain situations. Basically, it allows you to visualise orbits in terms of the standard constants of motion $(E, \mu, P_\varphi, \pm 1)$ but there is a catch. You have to interactively specify the magnetic field via some parameters. These parameters will create a solution to the Solov'ev equation for a magnetic equilibrium. The input variables include

- **m** - The mass of the ion for which the orbit is computed
- **q** - The charge of the ion for which the orbit is computed
- **Emin** - The minimum energy of the ion for which orbits are to be visualised
- **Emax** - The maximum energy of the ion for which orbits are to be visualised

If you have set reasonable input variable values, go ahead and start the app in the normal way. You should then be able to access the app via your web browser by going to the webpage 'localhost:port'. As per usual, it might take some

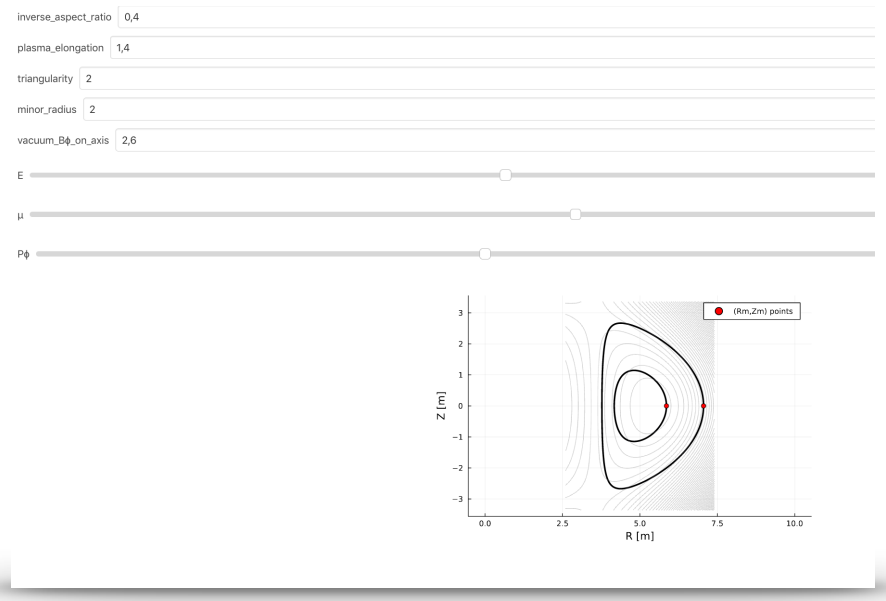
time to load. When it's done, you should see something similar to the screenshot below.



The app has quite the number of interactive parameters. Let's go through them one by one:

- **inverse_aspect_ratio** - This is the ratio of the small radius of the tokamak plasma, divided by the large radius of the tokamak plasma. That is a/R.
- **plasma_elongation** - This is the elongation of the plasma. Basically how elongated it is in the vertical direction compared to the horizontal direction.
- **triangularity** - The familiar triangularity δ of the tokamak plasma. Yes negative triangularity is possible.
- **minor_radius** - The minor radius of the plasma a. In meters.
- **vacuum_Bϕ_on_axis** - The strength of the magnetic equilibrium at the magnetic axis. In Teslas.

Then you have sliders for the three constants of motion. Notice that for some values, there will be two valid orbits as the screenshot below shows. This is to be expected from the standard $(E, \mu, P_{\varphi}, \pm 1)$ coordinates.



That concludes the chapter on the comWebApp.jl.

17.

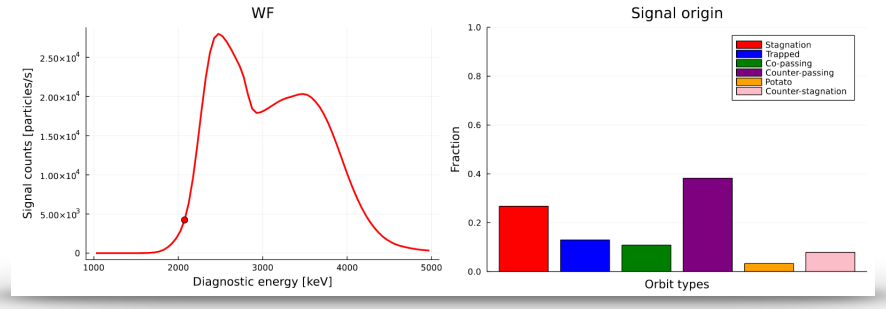
VISUALISING ORBIT SPLITS OF DIAGNOSTIC SIGNALS

- apps/signalWebApp.jl

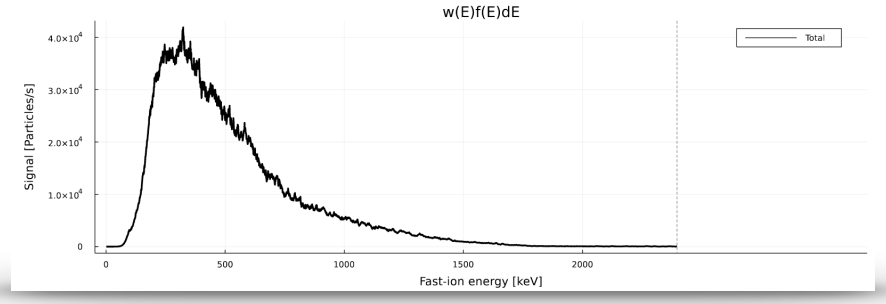
This app is arguably one of the flagships of the OWCF web applications. The inputs are few but the output plots are many. It can visualise orbits splits of diagnostic signals, in many forms. Let's find out what that means. When running the ps2WF.jl, did you set the *calcWFOs* input variable to true? If you did, you can use this app to visualise different signals in terms of orbit types. That is, you can split a diagnostic signal into its orbit types constituents. Let's go through the inputs

- **filepath_ps2WF_output** - This input is exactly what it sounds like. You specify the file path to the output file from the ps2WF.jl script. Please note! Keep in mind that the output file has to have been produced with the ps2WF.jl input variable *calcWFOs* set to true.
- **energy_ticks** - These will be the x-axis ticks of the $f(E)$ plot for example. You have to specify these manually because you are better at guessing these correctly than the algorithm.
- **pm_ticks** - These will be the x-axis ticks of the $f(p_m)$ for example. Same reason as for *energy_ticks.*
- **Rm_ticks** - These will be the x-axis ticks of the $f(R_m)$ for example. Same reason as for *pm_ticks.*

Now, let's start this wonder of an app. Once it has loaded, access it by opening your web browser and go to the website 'localhost:*port*. Wait for it to load, et voilà! You should see something similar to the screenshots below.

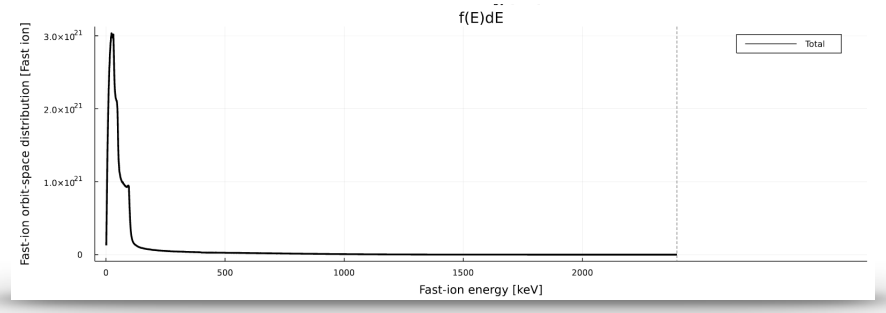


signalWebApp.jl - Signal orbit constituents

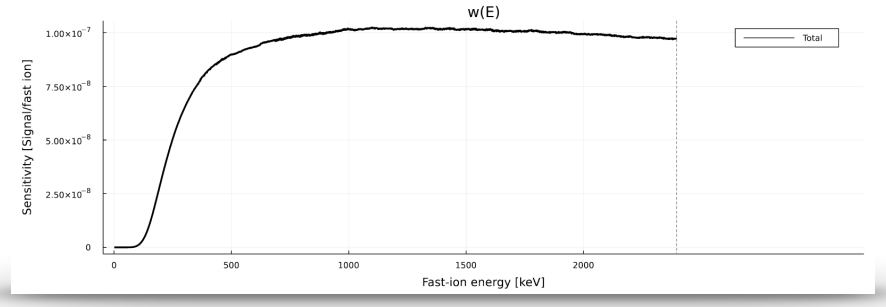


signalWebApp.jl - w(E)f(E) (total)

Then you also have the interactive buttons and sliders. With the 'Ed' slider, you can scan the available diagnostic energy bins. As you do so, you can see in the 'Signal origin' plot how the signal composition in terms of orbit types changes. You can also see how the 'w(E)f(E)' plot changes, to examine where the bulk of the signal comes from in terms of fast-ion energy. Similarly, the 'w(E)' plot will change and show



signalWebApp.jl - f(E) (total)



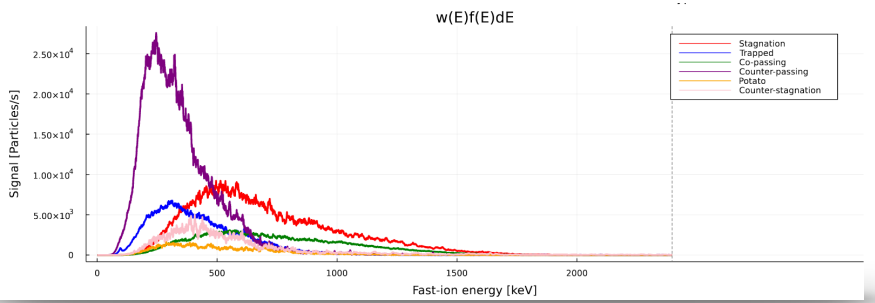
signalWebApp.jl - w(E) (total)

you what the fast-ion energy distribution of weights looks like for different diagnostic energy bins.

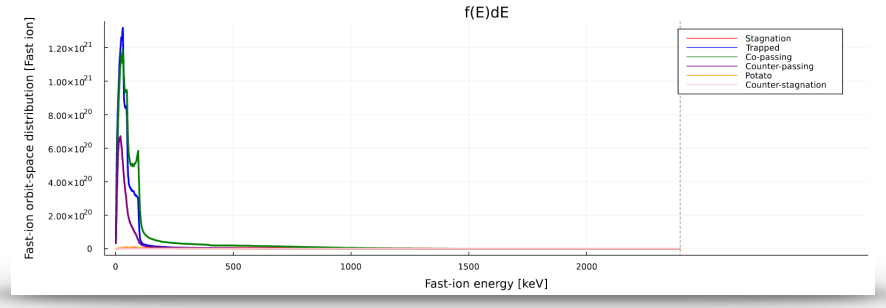
If you change the **density** app control from ‘total’ to ‘orbits’, all the plots in the ‘w(E)f(E)’, ‘f(E)’ and ‘w(E)’ figures will split into their orbit type constituents. As the screenshots below show.

Via this functionality, you can examine in detail where you can expect signal, fast-ion distribution and sensitivity in terms of orbit types. In the ‘signalWebApp.jl - w(E) (orbits)’ figure, we can for example observe how the sensitivity to counter-passing orbits decrease with fast-ion energy, while it increases for counter-stagnation orbits.

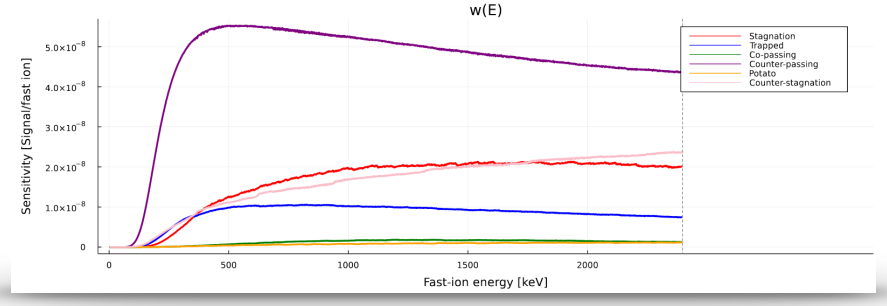
In the ‘signalWebApp.jl - w(E)f(E) (orbits)’ figure, we can observe how the counter-passing signal peaks around 250



signalWebApp.jl - w(E)f(E) (orbits)

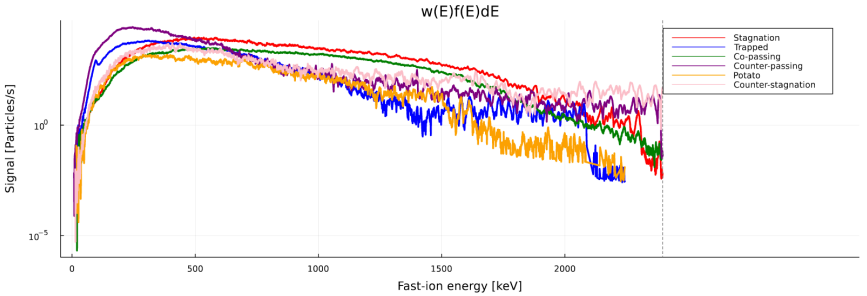


signalWebApp.jl - f(E) (orbits)

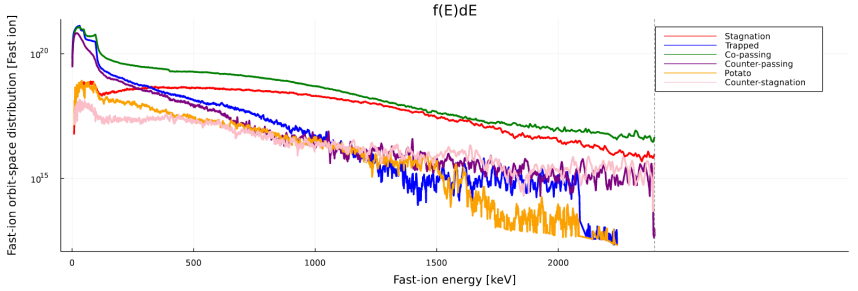


signalWebApp.jl - w(E) (orbits)

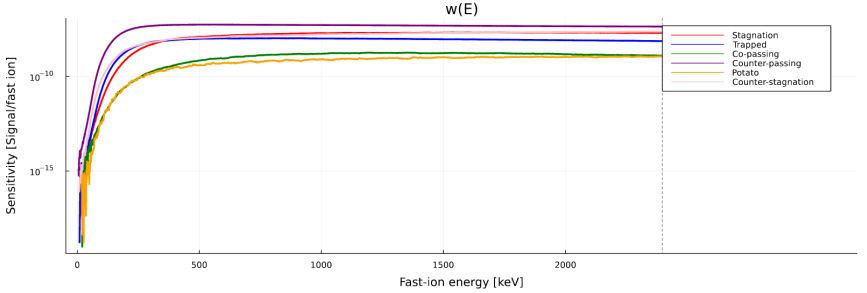
keV and then decreases. While the stagnation signal peaks at around 500 keV. However, since the ICRF tail is relatively small, we need to change the **y_scale** app control from ‘linear’ to ‘logarithmic’ to be able to deduce anything useful from the ‘f(E)’ plot. As the screenshots below show. That concludes the signalWebApp.jl chapter. Good luck, have fun!



signalWebApp.jl - w(E)f(E) (orbits)(log-y)



signalWebApp.jl - f(E) (orbits)(log-y)



signalWebApp.jl - w(E) (orbits)(log-y)

18.

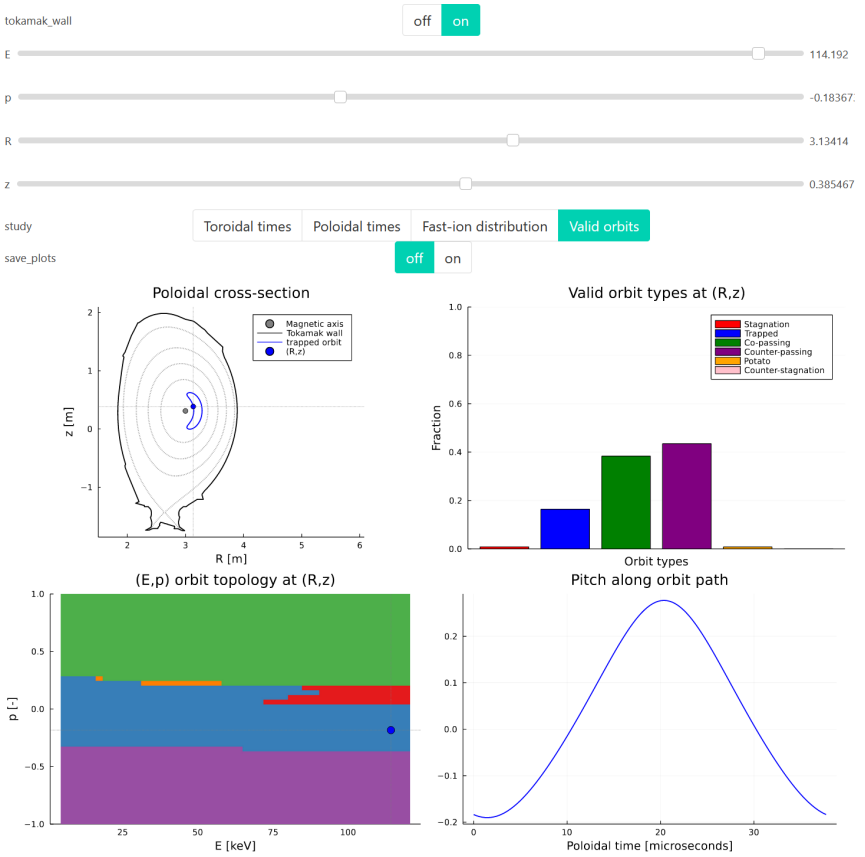
COMPUTING AND VISUALISING GUIDING-CENTRE TOPOLOGICAL MAPS

- calcEpRzTopoMap.jl
- apps/EpRzWebApp.jl

The OWCF also provides the utility to compute topological maps in (E, p, R, z) coordinates (as well as maps of poloidal and toroidal transit times). This can be done with the **calcEpRzTopoMap.jl** script. The results can then be interactively visualised via the **EpRzWebApp.jl**. Let's start with how to compute such 4D maps. As per usual, we copy the *start_calcEpRzTopoMap_template.jl* from the OWCF/ templates/ folder into the OWCF/ folder, renaming it as we do so. The inputs are the same as for the start_calcTopoMap_template.jl start file, except now for the (E, p, R, z) 4D space instead of the (E, p_m, R_m) 3D space. If you would like to be able to visualise maps of poloidal and toroidal transit times, please remember to set *saveTransitTimeMaps* to true. Once the calcEpRzTopoMap.jl script has finished, you should see an output in your *folderpath_o* folder that looks similar to the screenshot below.

Example of calcEpRzTopoMap.jl output file

Once you have the results file, go ahead and provide it as the input *filename_tm* to the EpRzWebApp.jl. Start the web application in the usual way (navigate to the OWCF/ folder in your terminal, start Julia, type 'include("apps/EpRzWebApp.jl")', wait for the 'Task (runnable)...' message, go to the 'localhost:port' website in your web browser et voilà!). Once it's loaded, you should see something similar to the screenshot below. Good luck using the EpRzWebApp.jl!



19.

COMPUTING A SOLOV'EV MAGNETIC EQUILIBRIUM

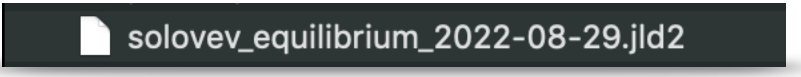
- extra/compSolovev.jl

The OWCF provides the possibility to create your own customisable magnetic equilibrium, instead of loading it from a TRANSP .eqdsk/.geqdsk input file. This is enabled via the solovev.jl utility of the Equilibrium.jl Julia package. The OWCF script extra/compSolovev.jl provides an easy-to-use tool for saving Solov'ev equilibria usable by other parts of the framework. The inputs include:

- **B0** - The magnetic field strength on axis.
- **R0** - The major radius position of the magnetic axis.
- ϵ - The inverse aspect ratio of the plasma.
- δ - The plasma triangularity.
- κ - The plasma elongation.
- α - The constant relating the beta regime.
- **qstar** - Kink safety factor.
- B0_dir - Specifying co- or counter-clockwise magnetic field direction when observing tokamak from above.
- Ip_dir - Specifying co- or counter-clockwise plasma current when observing tokamak from above.
- **diverted** - Do you want the tokamak to have a divertor?

Once all input variables have been specified, you run the compSolovev.jl script in the usual OWCF way: start Julia in the OWCF folder, type 'include("extra/compSolovev.jl")' and hit enter.

If all goes well, you should see a results file similar to the screenshot below in your *folderpath_o* output folder. You can then use that file as *filepath_equil* input variable in all OWCF start files that require one. That concludes the guide on how to use the compSolovev.jl OWCF script.



20.
EXTRACTING TOPOLOGICAL BOUNDARIES

- helper/extractTopoBounds.jl

The topological maps of the OWCF enable the user to distinguish between the different orbit types in the (E, p_m, R_m) and $(E, \mu, P_\varphi; \sigma)$ coordinate spaces. The *extractTopoBounds.jl* helper script provides the possibility to extract the phase-space coordinates of the boundaries between the topological regions. In practice, this is often done automatically when computing a topological map with the *calcTopoMap.jl* main script, by specifying the *includeExtractTopoBounds* input variable to true. The calcTopoMap.jl main script will then internally call the extractTopoBounds.jl helper script when a topological map has been computed. However, should the user wish to manually take a .jld2-file containing a topological map and extract the topological boundaries, this can be done by manually running the extractTopoBounds.jl helper script. The input variables include

- **partOfCalcTopoMap** - This should always be set to *true* by default. This is because the extractTopoBounds.jl helper script will then load the rest of the input variables from those already defined in the start file to calcTopoMap.jl. But if you are running the extractTopoBounds.jl helper script as a stand-alone script, be sure to set the partOfCalcTopoMap input variable to false. When it has completed and you are done with running it manually, **please make sure to reset it to true!**

- **filepath_tm** - The path to the .jld2-file which is the main output file of the calcTopoMap.jl main script

When all the input variables have been specified correctly, please go ahead and execute the extractTopoBounds.jl helper script in the usual OWCF way. When it has completed, you should see an output-file similar to the one in the screenshot below in the output folder specified by the *folderpath_o* input variable. That concludes the guide on how to use the *extractTopoBounds.jl* helper script.

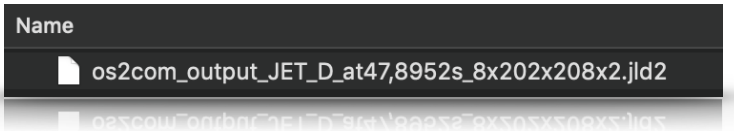
21.

MAPPING FROM ORBIT SPACE TO CONSTANTS-OF-MOTION SPACE

- helper/os2com.jl

All (E, p_m, R_m) orbit-space quantities that are computed with the OWCF, and that do not require a Jacobian, can be mapped to $(E, \mu, P_\phi; \sigma)$ constants-of-motion space. That can be done either 'in-app' (e.g. using the apps/weightsWebApp.jl and setting *enableCOM* to true) or via the helper/os2com.jl script.

The os2com script has a simple set of input variables. These include the *filepath_Q* input variable. This can be the filepath to an output from the orbWeights_2Dto4D.jl, the calcTopoMap.jl scripts etc. The important thing is that the format of the data quantity is either (E, p_m, R_m) or (E_d, E, p_m, R_m) where E_d are diagnostic energy bins of interest. Copy a start_os2com_template.jl file from the templates/ folder into the OWCF/ folder, modify the input variables and execute it via e.g. 'julia start_os2com_test.jl' (having changed the filename to start_os2com_test.jl when copying). The output file should then look similar to the screenshot below. That concludes this short chapter on how to use the os2com.jl helper script.



22.
THE BASIC DATA TYPES OF THE OWCF

As discussed in H. Järleblad *et al*, CPC, 2023, there are (as of version 1.0) five basic forms of data used by the OWCF. These include:

- A magnetic equilibrium
- A fast-ion distribution in (E,p,R,z) format
- A diagnostic sightline file, produced by the LINE21 code
- A TRANSP .cdf shot file
- A TRANSP .cdf fast-ion shot file

It should be clarified that these five forms of data are what the OWCF relies on as starting points for all its scripts. However, naturally, it produces other forms of data via executing of its scripts.

The magnetic equilibrium can be one out of two types. First, it may be provided as a .eqdsk/.geqdsk file. These type of files are used by TRANSP as magnetic equilibrium data input. For a breakdown of the storage structure of the .eqdsk/.geqdsk files, please see the *misc/eqdsk_file_breakdown.pdf* document. Second, the magnetic equilibrium may be specified as a Solov'ev equilibrium. This magnetic equilibrium can be fully customised via the extra/compSolovev.jl script.

A fast-ion distribution in (E,p,R,z) format can be specified in several ways. Either it can be specified as a .jld2 file, obtained for example via extra/getEpRzFIdistrFromTRANSP.jl. It can also be specified manually, as long as the necessary .jld2 file keys are included ("F_ps", "energy", "pitch", "R", "z"). Finally, it can also be specified as a .h5/.hdf5

file, as long as it includes the .h5 keys “f”, “energy”, “pitch”, “R” and “z”.

A diagnostic sightline file has to be specified as an output-file from the LINE21 code. It can also be left unspecified, whereupon the OWCF will assume 4*pi spherical emission.

A TRANSP .cdf shot file is quite self-explanatory. These files are usually named as “12345A01.cdf”. They contain a lot (!) of information about the TRANSP run with the corresponding ID number (ID 12345A01, in our example).

A TRANSP .cdf fast-ion shot file is an output file from the TRANSP NUBEAM module. It contains a lot (!) of information about the fast-ion distribution and shot info. It is usually named as “12345A01_fi_1.cdf”.

A.

APPENDIX - TWO-STEP REACTIONS

This version of the OWCF (v1.0) is heavily integrated with the DRESS code (J. Eriksson et al, CPC, 2016). Unfortunately, in this version of the OWCF and the DRESS code, two-step fusion reactions are not supported. That is, fusion reactions that occur in two steps, e.g. ^{9}Be(^{4}He, $n\gamma$)^{12}C

$$^{9}\text{Be} +^{4} \text{He} \rightarrow n +^{12} \text{C}^{*} \rightarrow n + \gamma +^{12} \text{C}$$

To make two-step fusion reactions possible to simulate with the OWCF, one would need to implement some sort of random step-function in the DRESS Python scripts fusreact.py, relkin.py and/or relscatt.py. That is, a random step is taken by e.g. the ^{12}C* excited carbon ion when born, and then it de-excites etc. The corresponding cross-sectional data for the ^{9}Be(^{4}He, $n\gamma$)^{12}C reaction would need to be implemented in fusreact.py.

B.
APPENDIX - .H5 BACKWARDS

As can be seen in the OWCF, at some points, the OWCF checks if the size of the fast-ion distribution data loaded from a .h5/.hdf5 file is consistent with the lengths of the $E-$, $p-$, $R-$ and $z-$grid vectors. If it is not, then it reverses the data (so that the first dimension becomes the last and the second dimension becomes the second to last, etc). This is because, sometimes, when the fast-ion distribution is saved as a 4D data array (e.g. with Python) in the .h5/.hdf5 file format, its dimensions are loaded automatically reversed when loaded by Julia.

The OWCF knows this can happen, and thus checks the size of the 4D data array containing the fast-ion distribution when loaded from a .h5/.hdf5 file, to ensure it is consistent with the lengths of the $E-$, $p-$, $R-$ and $z-$grid vectors. The user does not have to do anything but be aware of the problem. This issue is taken care of by the OWCF 'under the hood'.

C.
APPENDIX - INCOMPLETE ORBITS

Some guiding-center orbits are problematic to integrate. It could be that the orbit is a trapped orbit where the 'tips' of the banana-shaped poloidal projection of the orbit almost touch. If so, the fast ion will stay at the tips during an extensive amount of time, which might confuse the algorithm that integrates the equations of motion. It could also be that the orbit is simply 'at the wrong place, at the wrong time', i.e. that its trajectory goes through a place of magnetic ambiguity. This means that at some points, the integration algorithm might not know exactly where to evolve the trajectory, which results in stalling.

Regardless of how this stalling happens, it must be avoided. The efficiency of the OWCF relies on its ability to utilise the advanced integration algorithm in GuidingCenterOrbits.jl for computing guiding-center orbit trajectories on the order of 1 millisecond. If that cannot be ensured, the framework might stall for minutes, hours (or even, in some extreme cases, days!). The integration algorithm in GuidingCenterOrbits.jl works on two fronts. The initial approach of the algorithm is to try to integrate the equations of motion via adaptive integration. Adaptive integration essentially means that the step-size of the integration varies. This usually results in the most accurate orbit trajectory computed in the shortest amount of time. However, sometimes, this fails. It can be for a number of reasons, for example due to the reasons listed above. If the adaptive integration fails, the algorithm will instead try to integrate the equations of motion using a fixed step size. If

that also fails, it will use a ten times smaller step size, and try again. And again. And again. And... Meanwhile, the RAM usage starts to blow up because more and more points along the trajectory are being included. At the same time, the problematic point (e.g. the magnetic ambiguity, or the banana tips) is still there and the algorithm will not get past it, no matter how fine a step size it uses. In theory, it will get past it at some point. However, by then, the RAM usage and total computation time will have been blown completely out of proportion (e.g. hundreds of GB and hours, days).

To avoid this, the OWCF therefore simply tells the algorithm to disregard those orbits that fail to be successfully integrated via adaptive integration. Via non-organised empirical studies, it has been found that approximately one in every couple of hundred thousand orbits is such a problematic orbit. Such orbits will be labelled ‘incomplete’ by the OWCF, in accordance with how they are labelled by the GuidingCenterOrbits.jl Julia package.

The OWCF has therefore chosen to leave approximately one in every couple of hundred thousand orbits as ‘incomplete’ rather than spend minutes, hours or possibly even days trying to integrate them. We think that it’s a small price to pay, and it keeps the average guiding-center integration time to approximately 1 millisecond.

D.

APPENDIX - ACCESSING THE OWCF APPS REMOTELY

Yes. It is possible to start an OWCF app on one computer and access it from another. If you are on the same network (e.g. office network) or if you, for some reason, have completely disabled your firewalls, you can utilise an app remotely. To do so, start the OWCF in the normal way on one computer. Make sure that you note the port number and the (network or public) IP address. Then, on your second computer, open a web browser of your choice an go to the webpage “IP ADDRESS:port”. You should then be able to use the OWCF app on your second computer, even though its running on your first computer.

E.

APPENDIX - SLURM RAM-USAGE INACCURACY

Sometimes, on some computational clusters, the RAM-memory usage statistics in the emails sent out by SLURM will be inaccurate. It could for example be that it says “0.73 MB RAM-memory used” even though you know that loading the Julia packages alone would have required hundreds of MB. Why does this happen? It is likely a communication error between SLURM and the ClusterManagers.jl package. However, it might not happen on all SLURM clusters. Nevertheless, it is just something for you to keep in mind, as you examine the post-computation batch job statistics.

F.

APPENDIX - THE ADVANTAGE OF RETURNING INCOMPLETE ORBITS (CONTINUED)

For many scripts and apps in the OWCF, the orbit computation algorithm will use a couple of default keyword arguments when integrating the (relativistic) equations of motion. These include the action of returning so-called incomplete orbits when the adaptive integration algorithm fails. This is done instead of switching to integration with a fixed (time) step size. Incomplete orbits are simply a way for the OWCF to label orbits that never finished integration.

There are many advantages with this default behaviour. First, integration with a fixed (time) step size is (generally) much slower than adaptive integration. The OWCF needs to integrate thousands upon thousands of fast-ion orbits very quickly. If fixed (time) step size integration would be included as default behaviour, this would substantially slow down the framework by default.

Second, when the orbit integration algorithm fails to integrate the equations of motion after having tried both adaptive and fixed-step-size integration, it will try fixed-step-size integration again but with a 10 times smaller step size. This action of dividing the step size by a factor of 10 will be repeated if the algorithm keeps failing. Eventually, the integration will be so slow, and the RAM-memory usage so great that the program basically stalls. Or crashes due to

shortage of RAM-memory. Such a scenario should be avoided at all times.

Third, via empirical studies, it was found that approximately 1 in every 200 000 fast-ion orbits were 'problematic' in the sense that they seemed impossible to integrate. This could be because their (E, p, R, z) starting coordinate happened to be right in the middle of a heavily trapped orbit (also known as a pinch orbit. The inner leg effectively forms a counter-passing orbit, and the other leg effectively forms a co-passing orbit). In this extreme scenario, the particle might not move for a very (!) long time, thus making the integration algorithm stall. It could also be that the particle was initiated in a 'magnetically problematic point'. Either way, since these problematic orbits only occur approximately every 200 000th time, the choice to prioritise the speed of the OWCF is made.

For comparison, with the default choice to label these problematic orbits as incomplete, the OWCF (powered by the GuidingCenterOrbits.jl Julia package designed and originally written by Luke Stagner) can compute 200 000 guiding-center orbits in approximately 5 minutes. If instead the fallback to fixed-step-size integration would be used, it might instead take an hour or more to compute the same number of orbits.