ingrid

Bryan Max Garcia

May 12, 2021

CONTENTS:

INGRID (**IN**teractive **GRID**) is a Python based tokamak edge plasma grid generator capable of automatic generation of grids for magnetic-topologies with up to two x-points anywhere in the domain. The code can be operated in both a GUI mode and within scripts.

This documentation will get a user up and running with operating INGRID in GUI mode. For operating INGRID via scripts, API documentation has been generated (see Module Documentation).

The INGRID code is maintained by LLNL PLS-FESP. Issues with the code be brought to the attention of the current maintainers Bryan Garcia (UCSC), Maxim Umansky (LLNL), and Jerome Guterl (GA).

CHAPTER

ONE

INGRID INTRODUCTION

INGRID is an interactive grid generator for tokamak edge-plasma modeling capable of handling magnetic-configurations with two x-points in the computational domain.

INGRID analyzes EFIT data in order to classify the magnetic-topology, and generate the appropriate `gridue` formatted file for use in simulation code UEDGE.

Key features of INGRID include:

1. Support for single-null, unbalanced double-null, and snowflake (15, 45, 75, 105, 135, 165) configurations
2. Python based code with GUI and scripting usability
3. UEDGE friendly
4. Portable YAML parameter file driver
5. Modular design pattern for continued development efforts

INGRID was developed at LLNL PLS-FESP by Bryan Garcia (UCSC), Maxim Umansky (LLNL), and Jerome Guterl (GA).

CHAPTER

TWO

GETTING STARTED

In this section, we will explain how to install INGRID and walk you through an example case that generates a grid for a single-null configuration. Examples will be showcasing the GUI operation of INGRID.

2.1 Downloading and installing INGRID

2.1.1 Requirements

To run INGRID on your machine, `anaconda3` and `setuptools` must be installed and up to date. `anaconda3` installers can be found here.

Tip: You can create a new conda environment with the command `conda create --name myenv` (replace `myenv` with the environment name).

Once the Anaconda package manager is installed, `setuptools` can be added to the conda environment by running:

```
conda install -c anaconda setuptools
```

To update `setuptools` run:

```
pip install setuptools --upgrade
```

2.1.2 Obtaining the code

Clone the INGRID repo with the command:

```
git clone username@https://github.com/LLNL/INGRID.git IngridDir
```

where `IngridDir` is the name of the destination clone directory.

2.1.3 Installing INGRID

Warning: Users **not** on MacOS Mojave may skip this warning. **Read on otherwise**. MacOS Mojave has issues with certain backend libraries used in INGRID. These issues have been documented by Apple. As a workaround, a specific Conda evironment has been created and must be installed by Mojave users. Navigate into the cloned repo locate the file `conda_env.yml`. Create the mentioned Conda environment by running `conda env create -f conda_env.yml`. Activate the new Conda environment by running `conda activate ingrid`. When active, the terminal prompt should begin with `(ingrid)`. The `ingrid` Conda environment must be active for the next section.

The user will install INGRID with the `setup.py` file provided in the cloned repo. Installation begins by running:

```
python setup.py install --user
```

2.1.4 Contents

Within the cloned repo are a variety of directories containing source-code, drivers, example/template files for controlling INGRID (will be discussed later), and data that the provided example-files/demos use.

We will be utilizing the directory `example_files` in our tutorials, and we encourage you to utilize the items in directory `template_files` for your own INGRID usage.

2.2 Launching the INGRID GUI

Now that INGRID has been installed on the machine, you can now begin utilizing INGRID in both GUI mode and as an importable module in Python. We will focus on using INGRID's stand-alone GUI for now.

Warning: As a reminder, MacOS Mojave users must ensure the conda environment provided with the INGRID code has been installed. The following information assumes the user has done so (see *Downloading and installing INGRID* if unsure).

2.2.1 Launching from `drivers`

Although INGRID can be imported and launched from a Python session, we will first explain how to launch INGRID from the cloned repo.

We recommend new users launch INGRID in this way since the `example_files` we will explore in the tutorials come pre-set with relative paths to necessary data (eqdsk files, geometry files, etc…).

Note: Once the basic controls for setting paths to data withing the INGRID parameter file is understood, the user should be able to utilize the provided example-files without starting from the `drivers` directory.

To launch via the driver script, navigate into the cloned repository that we used to install INGRID. From here, navigate into the directory `drivers` and run the command:

```
python StartGUI.py
```

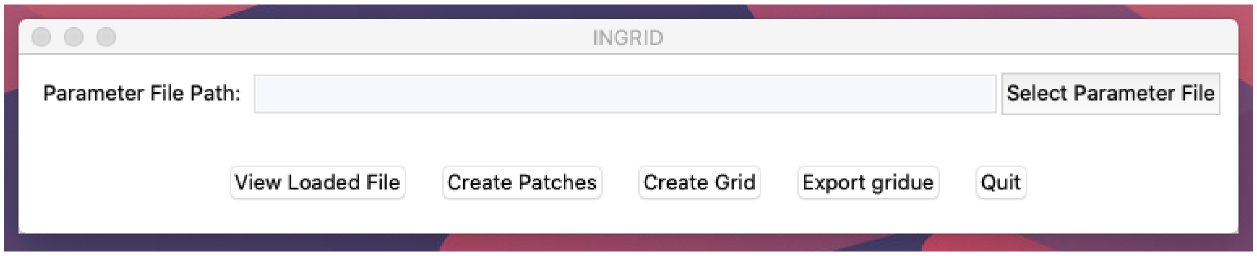
The INGRID GUI should now be visible, and ready for use.

2.2.2 Launching from a Python session

For users who know do not intend on using the YAML files within `example_files`, or those who understand how to set paths within the INGRID parameter file, you can launch the INGRID GUI from a Python prompt as follows:

```
>>> from INGRID import ingrid
>>> ingrid.QuickStart()
```

After executing the above commands, the INGRID GUI should be visible and ready for use (seen below).



Note: Upon launching the INGRID GUI, the user will be prompted with a plenty of terminal output. This is nothing to be alarmed of since this part the `Ingrid` class' initialization. As we load parameter files and modify settings, we should see appropriate changes reflected in the terminal output.

2.3 The INGRID parameter file

2.3.1 Background

INGRID has been designed to be controlled from a single configuration/parameter file when operating in GUI mode. This specially formatted YAML file is similar to Fortran namelist files due to the *key-value* structure it contains.

Here is a snippet of what a YAML formatted file can look like.

```
# Comments are supported and follow python convention!
# YAML entries are a mapping of the form:
#
#   key: value
#
# It follows that python interprets the YAML file as a dict

my_YAML_entry:
    # YAML file use spaces as indentation.
    # 2 or 4 spaces (pick one and stick to it) indicate a nested item.
    # Here we use 4 spaces.
    my_str_key: 'Hello, YAML!'  # my_str_key = 'Hello, YAML!'
    my_int_key: 32  # my_int_key = 32
    some_float: 3.14159  # some_float = 3.14159
    my_true_bool: true # bool case-insensitive
    my_false_bool: 0  # '1' '0' bool values supported.

    # Empty lines within file are ok.

```

(continues on next page)

(continued from previous page)

```
    # Just remember it's the spaces that matter.

    my_sub_dict: # Nested dicts are supported.
        another_key: ending my example code-block here.
        # The above value entry will be interpreted as a string
        # (no quotations needed)
```

Although the INGRID parameter file contains many controls for a user, it does not stray from the patterns illustrated above (no advanced YAML knowledge required).

Tip: While operating INGRID in GUI mode, keep your favorite text-editor handy with the parameter-file in use loaded. You will be making frequent edits to this core parameter/configuration file.

Although not necessary for following tutorial, detailed documentation of the INGRID parameter-file can be found here.

2.3.2 A single-null example file

Users of INGRID have plenty of controls available for fine-tuning their final grid. This section explains how to navigate some key controls within the INGRID parameter file.

We will be using the pre-populated INGRID parameter file `DIIID_SNL.yml` from the `example_files` directory in the single-null walk-through. Open this file in your preferred text-editor. At the time of writing this documentation, the parameter file `DIIID_SNL.yml` contains the following:

```
# ---------------------------------------------------
# User data directories
# ---------------------------------------------------
dir_settings:
  eqdsk: ../data/SNL/DIII-D/  # dir containing eqdsk
  limiter: .  # dir containing limiter
  patch_data: ../data/SNL/DIII-D/  # dir containing patch data
  target_plates: ../data/SNL/DIII-D/ # dir containing target plates

# ---------------------------------------------------
# eqdsk file name
# ---------------------------------------------------
eqdsk: neqdsk

# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # --------------------------------------------------------------------------
  # Settings for grid generation (num cells, transforms, distortion_correction)
  # --------------------------------------------------------------------------
  grid_generation:
    distortion_correction:
      all:
        active: True # true, 1 also valid.
        resolution: 1000
        theta_max: 120.0
```

(continues on next page)

(continued from previous page)

```
      theta_min: 80.0
    np_default: 3
    nr_default: 3
    poloidal_f_default: x, x
    radial_f_default: x, x
  # ---------------------------------------------------
  # guard cell size
  # ---------------------------------------------------
  guard_cell_eps: 0.001
  # ---------------------------------------------------
  # num levels in efit plot
  # ---------------------------------------------------
  nlevs: 30
  # ---------------------------------------------------
  # num xpts
  # ---------------------------------------------------
  num_xpt: 1
  patch_generation:
    strike_pt_loc: target_plates # 'limiter' or 'target_plates'
    rmagx_shift: 0.0
    zmagx_shift: 0.0
  # ---------------------------------------------------
  # Psi levels
  # ---------------------------------------------------
  psi_1: 1.066
  psi_core: 0.95
  psi_pf_1: 0.975
  # ---------------------------------------------------
  # magx coordinates
  # ---------------------------------------------------
  rmagx: 1.75785604
  zmagx: -0.0292478683
  # ---------------------------------------------------
  # xpt coordinates
  # ---------------------------------------------------
  rxpt: 1.300094032687
  zxpt: -1.133159375302
  # ---------------------------------------------------
  # Filled contours vs contour lines
  # ---------------------------------------------------
  view_mode: filled

# ---------------------------------------------------
# Saved patch settings
# ---------------------------------------------------
patch_data:
  file: LSN_patches_1597099640.npy
  preferences:
    new_file: true
    new_fname: LSN_patches_1597099640.npy
  use_file: false
```

(continues on next page)

(continued from previous page)

```
# ---------------------------------------------------
# Integrator
# ---------------------------------------------------
integrator_settings:
  dt: 0.01
  eps: 5.0e-06
  first_step: 5.0e-05
  max_step: 0.064
  step_ratio: 0.02
  tol: 0.005

# ---------------------------------------------------
# Limiter settings
# ---------------------------------------------------
limiter:
  file: ''
  use_efit_bounds: false

# ---------------------------------------------------
# target plate settings
# ---------------------------------------------------
target_plates:
  plate_E1:
    file: d3d_otp.txt
    zshift: -1.6
  plate_W1:
    file: d3d_itp.txt
    zshift: -1.6
```

Let's highlight some important entries that are often used when operating INGRID for single-null cases (basic usage). Advanced tutorials will also be provided.

2.3.3 Setting data paths

A user can provide a string that indicates the path to certain data. This is used to tell INGRID where to look for EFIT data, target plate coordinate, limiter coordinates, and patch-data (for reconstruction). We can set these paths by editing the entry:

```
# ---------------------------------------------------
# User data directories
# ---------------------------------------------------
dir_settings:
  eqdsk: ../data/SNL/DIII-D/  # dir containing eqdsk
  limiter: .  # dir containing limiter
  patch_data: ../data/SNL/DIII-D/  # dir containing patch data
  target_plates: ../data/SNL/DIII-D/ # dir containing target plates
```

Note: INGRID supports both absolute paths and paths relative to where INGRID has been launched.

If `dir_settings` is missing any entries, INGRID will (internally) set the missing values to a default value of `'.'` (current working directory). This holds even if `dir_settings` is omitted from the parameter file.

Note: `dir_settings` entries are the **directory** to look for data and NOT the file itself.

2.3.4 Providing an EQDSK file

The user provides the actual EQDSK file name separate from the `dir_settings` entry. We provide this at the global YAML level under entry `eqdsk`. That is:

```
# ---------------------------------------------------
# eqdsk file name
# ---------------------------------------------------
eqdsk: neqdsk
```

Note: In this example, INGRID searches for the file `neqdsk` within the directory `../data/SNL/DIII-D/` (relative to the launch point) since `dir_settings['eqdsk']` was set to `../data/SNL/DIII-D/` (see above).

2.3.5 Defining target plates

All target plate settings are under the global INGRID parameter file entry `target_plates`. We see this as:

```
# ---------------------------------------------------
# target plate settings
# ---------------------------------------------------
target_plates:
  plate_E1:
    file: d3d_otp.txt
    zshift: -1.6
  plate_W1:
    file: d3d_itp.txt
    zshift: -1.6
```

INGRID adopts a N-S-E-W compass direction notation in order to help generalize and simplify grid generation. It is important for a user to eventually learn these conventions. A detailed discussion of INGRID's naming conventions can be found here.

For now (in the case of a lower single-null configuration), note that entries `plate_E1` and `plate_W1` correspond to the *outer* and *inner* target plates, respectively. Each plate entry recognizes sub-entries `file` (file name to load), `zshift` (z-translation) and `rshift` (r-translation, not utilized and internal to INGRID defaults to `0.0`).

2.3.6 Defining x-points, magnetic-axis, and psi-levels

Settings for x-point coordinates, magnetic-axis coordinates, and psi-levels are found under the global INGRID parameter file entry `grid_settings`.

```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # ...
```

(continues on next page)

(continued from previous page)

```
# ...
# Other items currently not of interest...
# ...
# ...

# ---------------------------------------------------
# num xpts
# ---------------------------------------------------
num_xpt: 1

# ---------------------------------------------------
# Psi levels
# ---------------------------------------------------
psi_1: 1.066  # SOL
psi_core: 0.95  # CORE
psi_pf_1: 0.975  # PRIVATE-FLUX

# ---------------------------------------------------
# magx coordinates
# ---------------------------------------------------
rmagx: 1.75785604
zmagx: -0.0292478683

# ---------------------------------------------------
# primary xpt coordinates
# ---------------------------------------------------
rxpt: 1.300094032687
zxpt: -1.133159375302
```

Warning: The entry `num_xpt` is one of the most important entries in the INGRID parameter file since it determines INGRID's method of analysis. Dealing with more than one x-point requires a more in-depth understanding of the parameter file, so ensure this is set to the correct number of x-points.

2.4 Example: single-null configuration (introduction)

Warning: This tutorial assumes INGRID has been launched from the provided GUI driver. See page *Launching the INGRID GUI* for explaination.

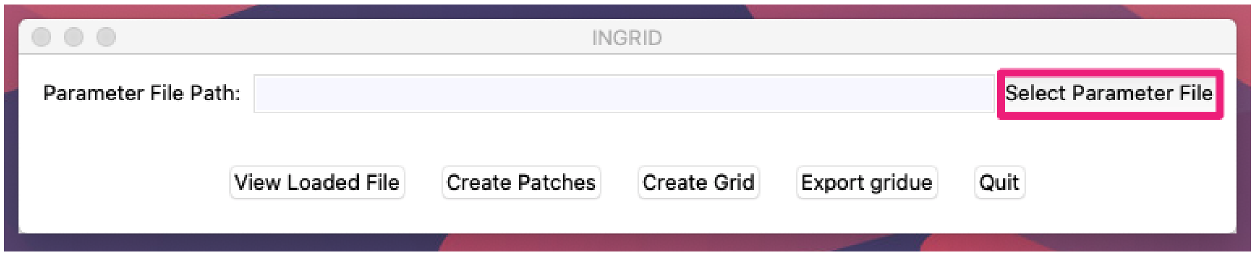
Here we will demonstrate how to generate a grid for a lower single-null (`SNL`) configuration. This tutorial aims to:

- Explain the GUI capabilities
- Illustrate the INGRID workflow (data analysis → patch-generation → grid-generation → exporting gridue)
- Expose the user to key parameter-file controls (see parameter-file documentation for further details)

Note: Although we are creating a lower single-null grid here, INGRID internally treats both lower and upper single-null configurations as `SNL` class instances. This means there is *no difference in user operation for generating a grid for* *either LSN or USN configurations.*

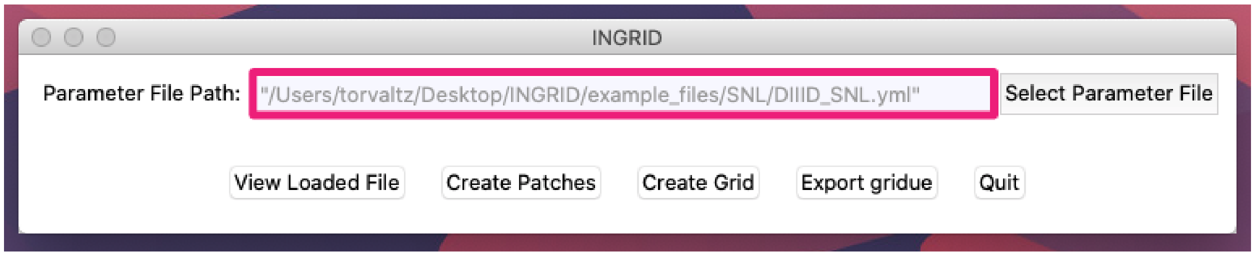
2.4.1 Loading our first example

After getting the INGRID GUI up and running, click the GUI button labeled “Select Parameter File” shown below boxed in red.



From here, your machine’s native file navigator should be on the screen. Navigate into the directory `example_files` the cloned repository to find a collection of example-cases we used for showcasing INGRID’s capabilities (as well as for testing the product).

Now navigate into directory `SNL` and select the file `DIIID_SNL.yml`. The GUI should now be updated with the loaded path to the example-file we selected as seen below boxed in red.



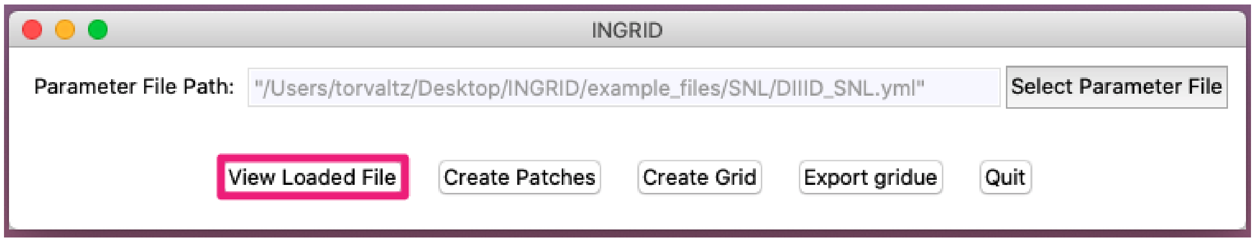
INGRID has now processed the selected parameter-file. Some (of many) actions executed automatically by INGRID include:

- Processing of paths to data (`EFIT`, geometry, patch-data, etc.)
- Loading of `EFIT` data
- Loading of strike-geometry (target plates and/or limiter)
- Refining of x-point coordinates
- Refining of magnetic-axis coordinates
- Initialization of visualization settings

With the data loaded, we can now proceed.

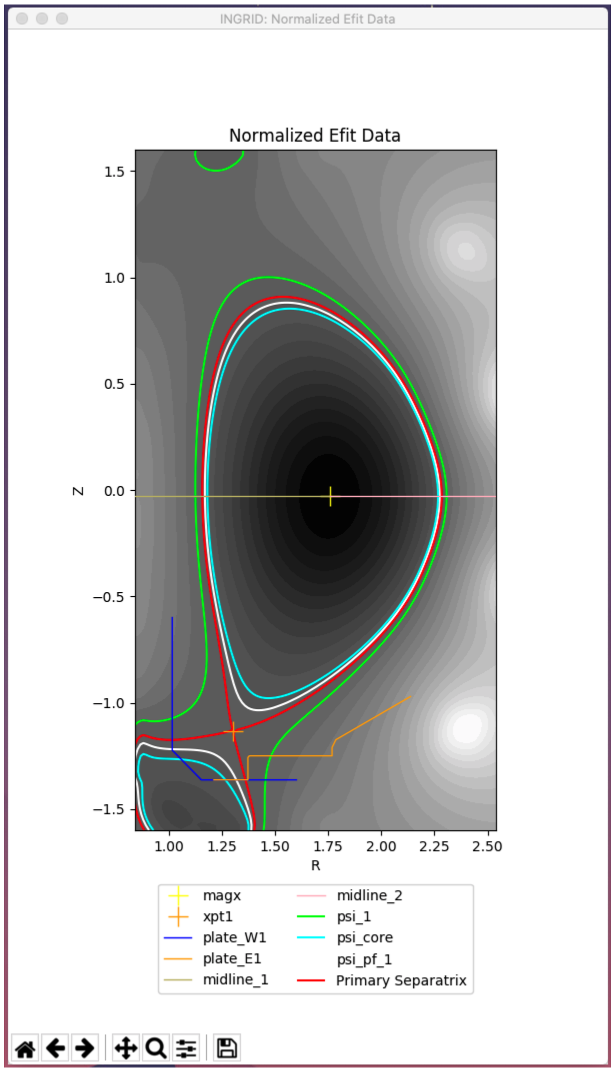
2.4.2 Viewing loaded data

To view the EFIT data, loaded strike-geometry, and psi-levels that will dictate our final grid, simply select "View Loaded File" shown below boxed in red.


INGRID

Parameter File Path: "/Users/torvaltz/Desktop/INGRID/example_files/SNL/DIIID_SNL.yml" Select Parameter File

Once clicked, we are greeted with a new plot window showing the DIII-D data we have loaded.



Here are some key items that INGRID has plotted (as seen in the legend):

- **Refined** primary x-point coordinate as an orange '+' marker (`xpt1`)
- **Refined** magnetic-axis coordinate as a yellow '+' marker (`magx`)
- **Normalized** `eqdsk` data as black and white filled contours (psi-value of 0 at `magx` and psi-value of 1 at `xpt1`)

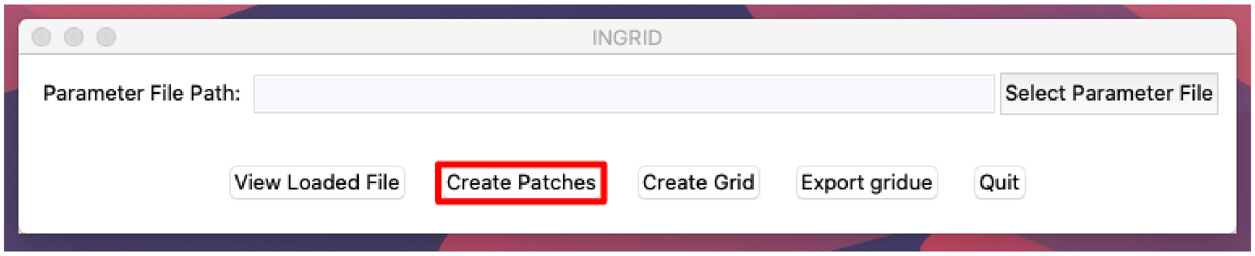
- `plate_W1` data as a dark blue line (LSN inner target plate)
- `plate_E1` data as an orange line (LSN outer target plate)
- The primary separatrix (red contour line)
- SOL boundary (lime contour line)
- CORE boundary (cyan contour line)
- PRIVATE-FLUX boundary (white contour line)

Note: The user provides the approximate primary x-point coordinates (`rxpt1`, `zxpt1`), and magnetic axis coordinates (`rmagx`, `zmagx`) in the INGRID parameter file. INGRID takes these as an initial guess to provide to a root-finder in order to **refine** the user-provided coordinates to high-accuracy. These values are used internally throughout the user-session.

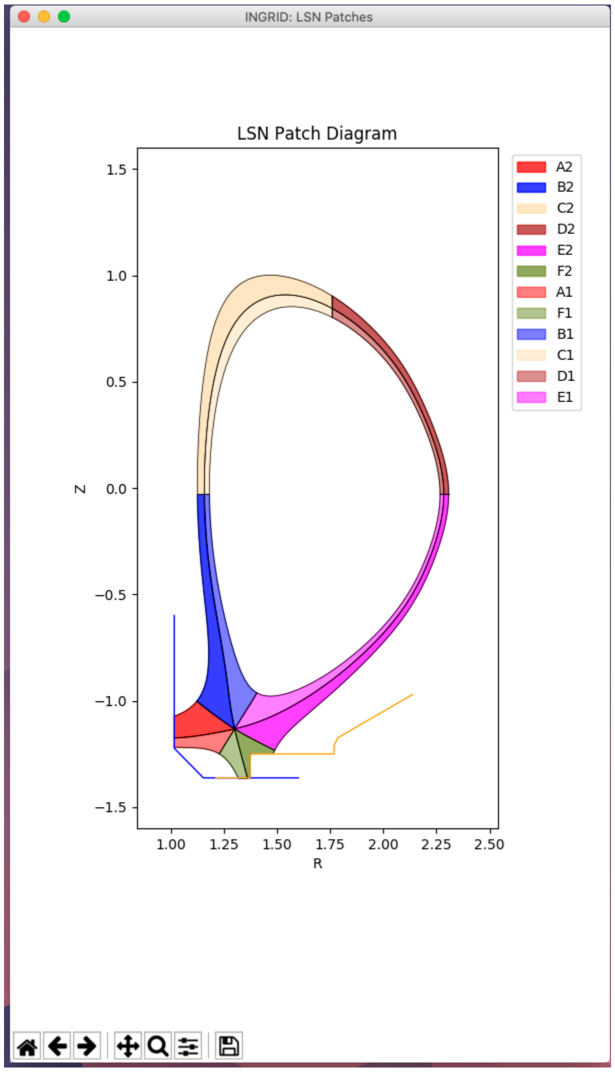
This stage is where the user will interact the most with the INGRID parameter file (tweaking psi-values, target-plate locations, limiter data, etc). Said settings will be used to generate the patch map we will see in the next section. Since these have already been provided for you, let us proceed to creating patches.

2.4.3 Creating patches

INGRID interally uses a geometry object hierarchy (`Point` ∈ `Line` ∈ `Patch` ∈ `TopologyUtils`) to generate the final gridue file. We will now create a collection of `Patch` objects. These `Patch` objects are quadrilaterals that form a *partition* of the region we are interested in generating a grid for. Before elaborating further, let us now create said collection of patches (referred to as a *Patch map*) by clicking the GUI button labeled "Create Patches" shown below boxed in red.



Once clicked, INGRID begins line-tracing in order to generate the Patch map seen below.



The collection of Patch objects are pictured in the Patch map. These Patch objects will generate their own subgrid that will be stitched together to form the exported global grid.

2.4.4 Saving Patch data

INGRID provides the user the capability of saving Patch data into a specially formatted NumPy npy files for later reconstruction. We control this feature within the parameter file by modifying the entries under patch_data (seen below):

```
# ---------------------------------------------------
# Saved patch settings
# ---------------------------------------------------
patch_data:
  file: LSN_patches_1597099640.npy
  preferences:
    new_file: true
    new_fname: LSN_patches_1597099640.npy
  use_file: false
```

Here we have:

- `file` - the name of the file to use for `Patch` reconstruction
- `preferences` - settings for configuring final `Patch` data file
  - `new_file` - create a new `Patch` data file
  - `new_fname` - name of new `Patch` data file
- `use_file` - use the provided file for `Patch` reconstruction

Note: Remember to set the directory to search for a `Patch` data *file* by modifying `patch_data` under entry `dir_settings`

Because the parameter file is populated with the above settings, we see that after creation of a `Patch` map the terminal prompts the user with a message stating:

```
# Saved patch data for file LSN_patches_1597099640.npy
```

The user is encouraged to try this feature out. To do so, first change the `use_file` entry within `patch_data` to a value of `True` and save the file. Now, close the `Patch` map window, and click `Create Patches` again. The `Patch` map should now be restored back to the state it was at.

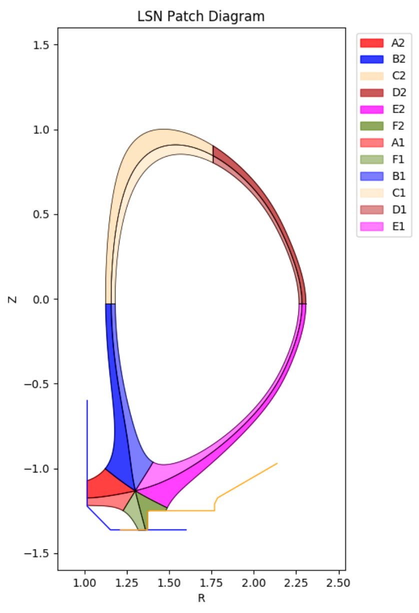
Tip: `Patch` data files expedite the grid generation process by bypassing all line-tracing. This feature is also useful for trading cases with other INGRID users

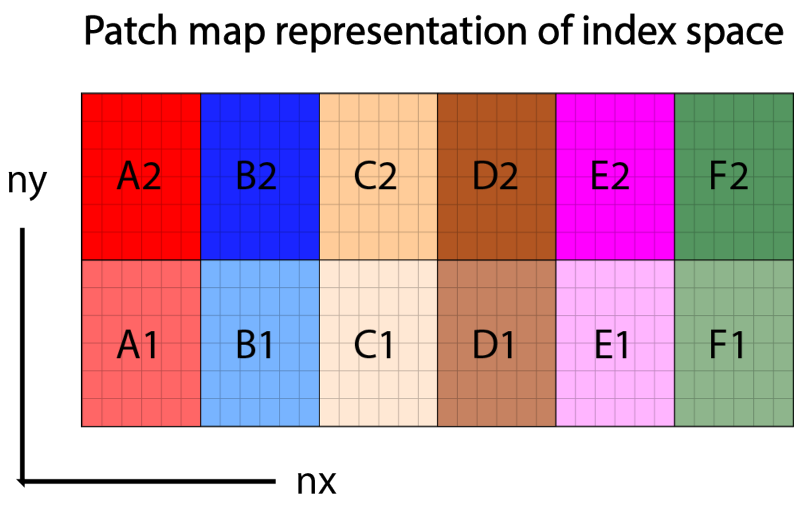
2.4.5 The `Patch` map

In the above plot we can see the "`Patch` map". Each `Patch` is been assigned it's own color, as well as a `Patch` label/tag consisting of a two-character string of the form "<alpha_char><numeric_char>"". This coding directly represents the index space of the final grid with:

- The alpha-char ("A", "B", . . . , "F" here) representing a poloidal "column" in the index space.
- The numeric-char ("1" and "2" here) representing a radial "row" in the index space.

Below is diagram illustrating said notation.





This notation proves to be robust since it holds for not only `SNL` topologies (both LSN and USN), but also all topologies such as `UDN` and the family `SF*`.

For the `SNL` family of configurations, the collection of `Patch` objects with numeric_char == "2" ("**A2**" - "**F2**") represent the SOL, `Patch` objects "**A1**" and "**F1**" represent the PF region, and `Patch` objects "**B1**", "**C1**", "**D1**", and "**E1**" represent the CORE.

Note: We will use this notation extensively for fine-tuning the final grid

Tip: `Patch` objects are ordered alphabetically clock-wise around the magnetic-axis and enumerated in direction of increasing psi

Now that we have partitioned the EFIT domain into the region we wish to model, let us now generate a grid.

2.4.6 Creating a grid

Before generating a grid, let's take a look at some grid controls in the INGRID parameter file.

Below are some entries we will be modifying.

```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # --------------------------------------------------------------------------
  # Settings for grid generation (num cells, transforms, distortion_correction)
  # --------------------------------------------------------------------------
  grid_generation:
    distortion_correction:
      all:
        active: True # true, 1 also valid.
```

(continues on next page)

(continued from previous page)

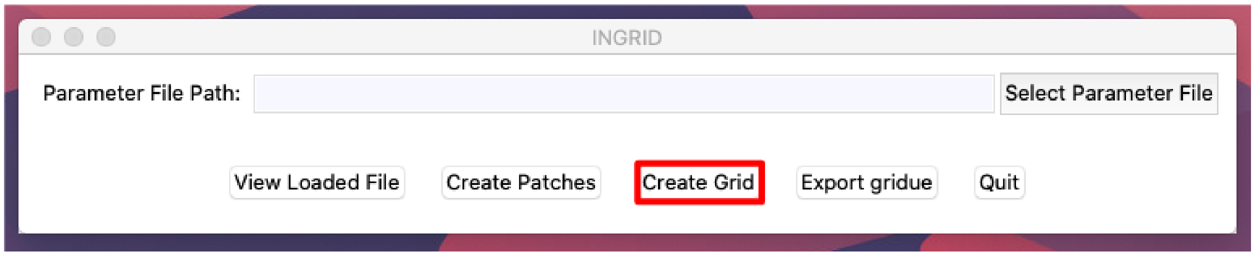
```
        resolution: 1000
        theta_max: 120.0
        theta_min: 80.0
    np_default: 3
    nr_default: 3
    # Other grid settings
```

Within the entry `grid_settings`, we have:

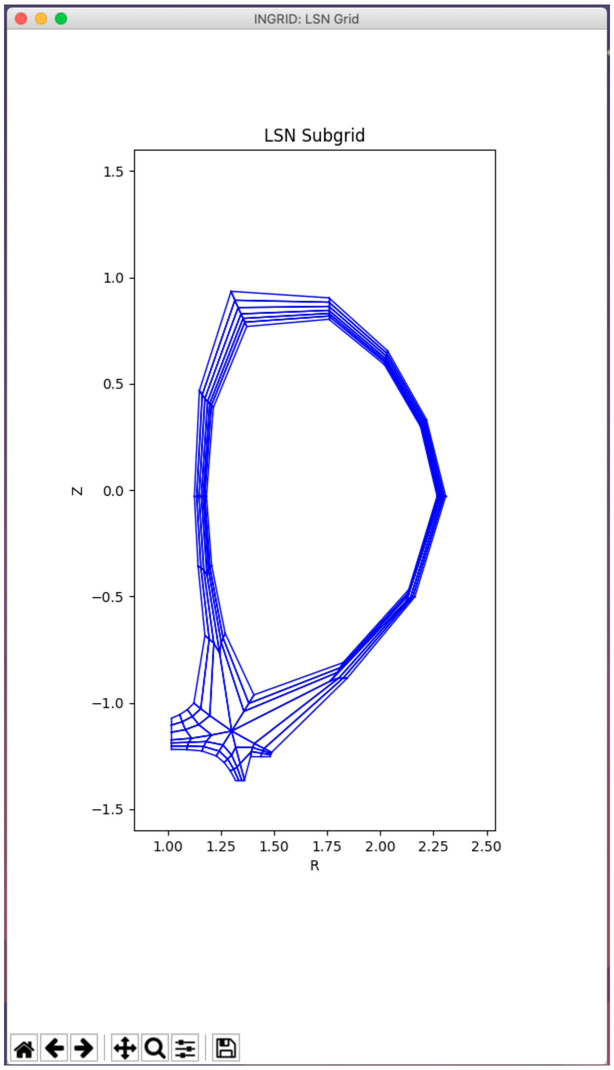
- `grid_generation` - settings for controlling resultant grid
  - `distortion_correction` - settings for controlling shearing in grid
  - `np_default` - default number of poloidal cells per `Patch`
  - `nr_default` - default number of radial cells per `Patch`

Note: We will work with entry `distortion_correction` at a later time (next section). For now, set the entry value to `False` so that we can see it's effects later

To execute refinement of the `Patch` map into a grid, we click the GUI button `Create Grid`.



The terminal will prompt the user with the progress of `Patch` refinement by providing a short summary of the subgrid being generated within each `Patch`. When `Patch` refinement has finalized, we are greeted with a new window showing the resultant grid.

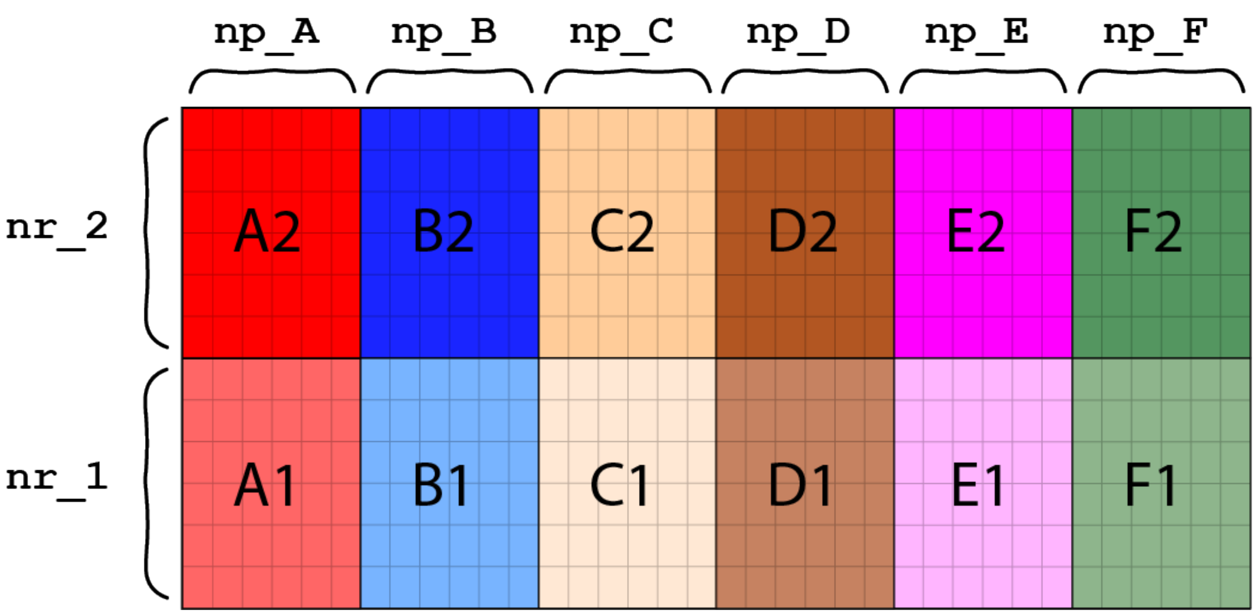


Although this grid can be immediately exported, there are still actions we can take to improve our grid naively generated with only `np_default` and `nr_default`.

2.4.7 Fine-tuning the grid

Generating grids with global values `np_default` and `nr_default` is not enough in many cases. INGRID allows users to specify the number of poloidal and radial cells for particular regions of the `Patch` map. This allows for refining the grid near regions of interest while maintaining global/default grid values per `Patch`.

To utilize this feature, we will fall back on the `Patch` naming convention explained in section *The Patch map*. The figure below shows a collection of keyword entries (`np_A`, `np_B`, … `np_F`, `nr_1`, `nr_2`) that can be added to the INGRID parameter file to control the number cells in a grid.



Note how modifying `np_A` would affect both “A2” and “A1” since they are poloidally dependent in index-space. Similarly, we see how modifying `np_1` would affect “A1” - “F1” since they are radially dependent in index-space.

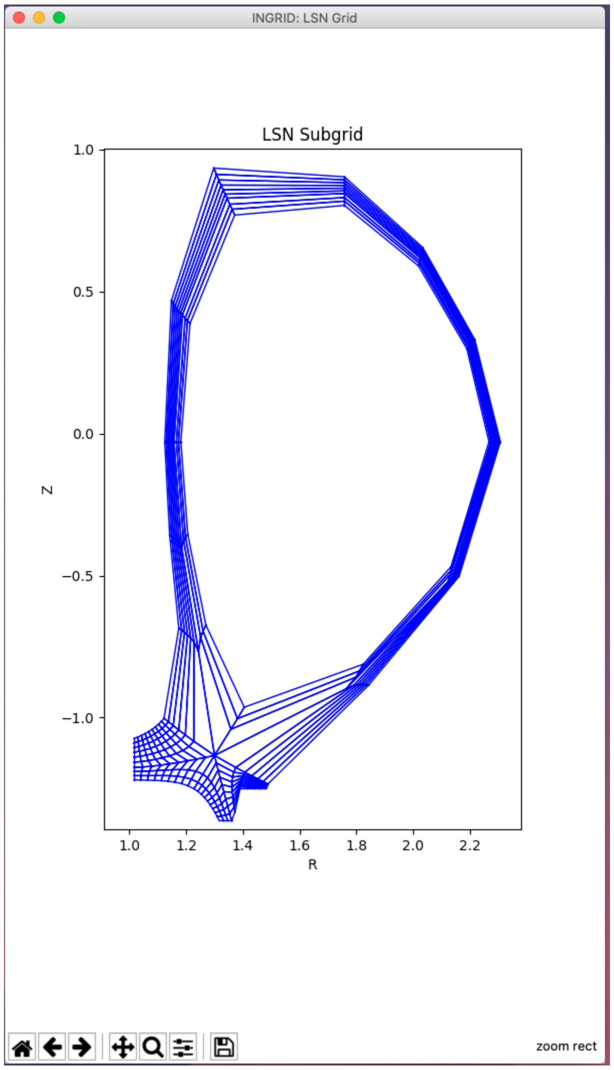
Let’s illustrate this idea by increasing the number of poloidal cells near both target plates. We see by inspecting the `Patch` map that target plates border patches “A*” and “F*”. This says we must add entries `np_A` and `np_F` to the INGRID parameter file. That is:

```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # ----------------------------------------------------------------------------
  # Settings for grid generation (num cells, transforms, distortion_correction)
  # ----------------------------------------------------------------------------
  grid_generation:
    distortion_correction:
      all:
        active: false # true, 1 also valid.
        resolution: 1000
        theta_max: 120.0
        theta_min: 80.0

    np_A: 9  # Create 9 poloidal cells in patches A1 and A2
    np_F: 9  # Create 9 poloidal cells in patches F1 and F2

    np_default: 3
    nr_default: 3
    # Other grid settings
```

In addition to refining poloidally, let’s increase the radial resolution near the target plates. In this case “A2” and “A1” are *not* dependent on each other (as seen in figure above). On the other hand, since the SOL consists of all patches with numeric-tag “2”, modifying “A2” in radial cells will modify all other patches in the SOL radially to keep the index-map consisitent. We will choose to refine “A2”. That is:

```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # ----------------------------------------------------------------------------
  # Settings for grid generation (num cells, transforms, distortion_correction)
  # ----------------------------------------------------------------------------
  grid_generation:
    distortion_correction:
      all:
        active: false # true, 1 also valid.
        resolution: 1000
        theta_max: 120.0
        theta_min: 80.0

    np_A: 9  # Create 9 poloidal cells in patches A1 and A2
    np_F: 9  # Create 9 poloidal cells in patches F1 and F2

    nr_2: 6  # Create 6 radial cells in layer 2

    np_default: 3
    nr_default: 3
    # Other grid settings
```

After making the addition, save the file and click "Create Grid". INGRID will detect that an edit was made to the parameter file and apply all changes. When `Patch` refinement has finalized, we are greeted with a new window showing the updated grid.



We can continue to modify the grid in order to allocate more cells near the x-point. A natural choice would be to target `np_B` and `np_E`. Doing so with the values `np_B = 18` and `np_E = 18` (double resolution for the larger patches), we see our parameter file consists of:

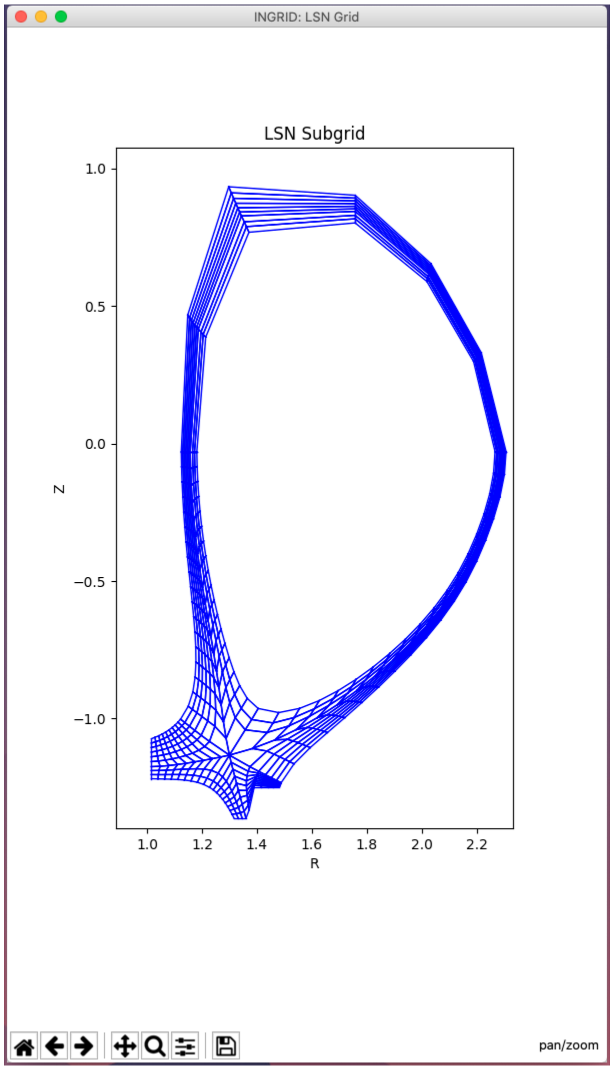
```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
  # ---------------------------------------------------------------------------
  # Settings for grid generation (num cells, transforms, distortion_correction)
  # ---------------------------------------------------------------------------
  grid_generation:
    distortion_correction:
      all:
        active: false # true, 1 also valid.
        resolution: 1000
        theta_max: 120.0
        theta_min: 80.0
```

(continues on next page)

(continued from previous page)

```
    np_A: 9  # Create 9 poloidal cells in patches A1 and A2
    np_F: 9  # Create 9 poloidal cells in patches F1 and F2
    np_B: 18  # Create 18 poloidal cells in patches B1 and B2
    np_E: 18  # Create 18 poloidal cells in patches E1 and E2

    nr_2: 6  # Create 6 radial cells in layer 2

    np_default: 3
    nr_default: 3
    # Other grid settings
```

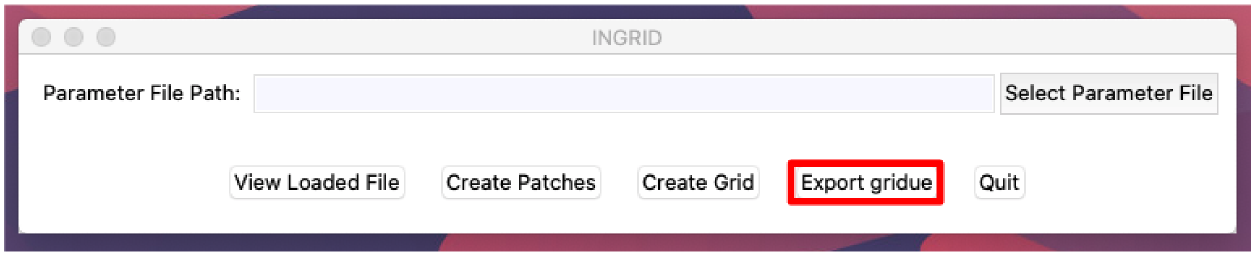
and produces a grid that we can see below (zoomed with Matplotlib toolbar provided in plots).



For the purposes of this introductory tutorial, let us continue to exporting the `gridue` file.

2.4.8 Exporting a `gridue` file

When the user is satisfied with the generated grid, a `gridue` formatted file can be generated by selecting "Export gridue" shown below boxed in red.



From here, the user will be able to select a save location for their INGRID generated `gridue` file.

2.4.9 Summary

In this tutorial, we demonstrated how to generate a `gridue` file for an `SNL` configuration. This introductory tutorial is not an exhaustive demonstration of INGRID's capabilities for grid generation. Other capabilities such as customizing the `Patch` map, applying poloidal/radial grid transformations, and mitigating cell-shearing can be found in the next `SNL` example case.

2.5 Example: single-null configuration (further exploration)

Note: This tutorial assumes the reader has already explored the introductory SNL tutorial.

Some cases **require** enabling of certain attributes in the parameter file in order to successfully produce a grid.

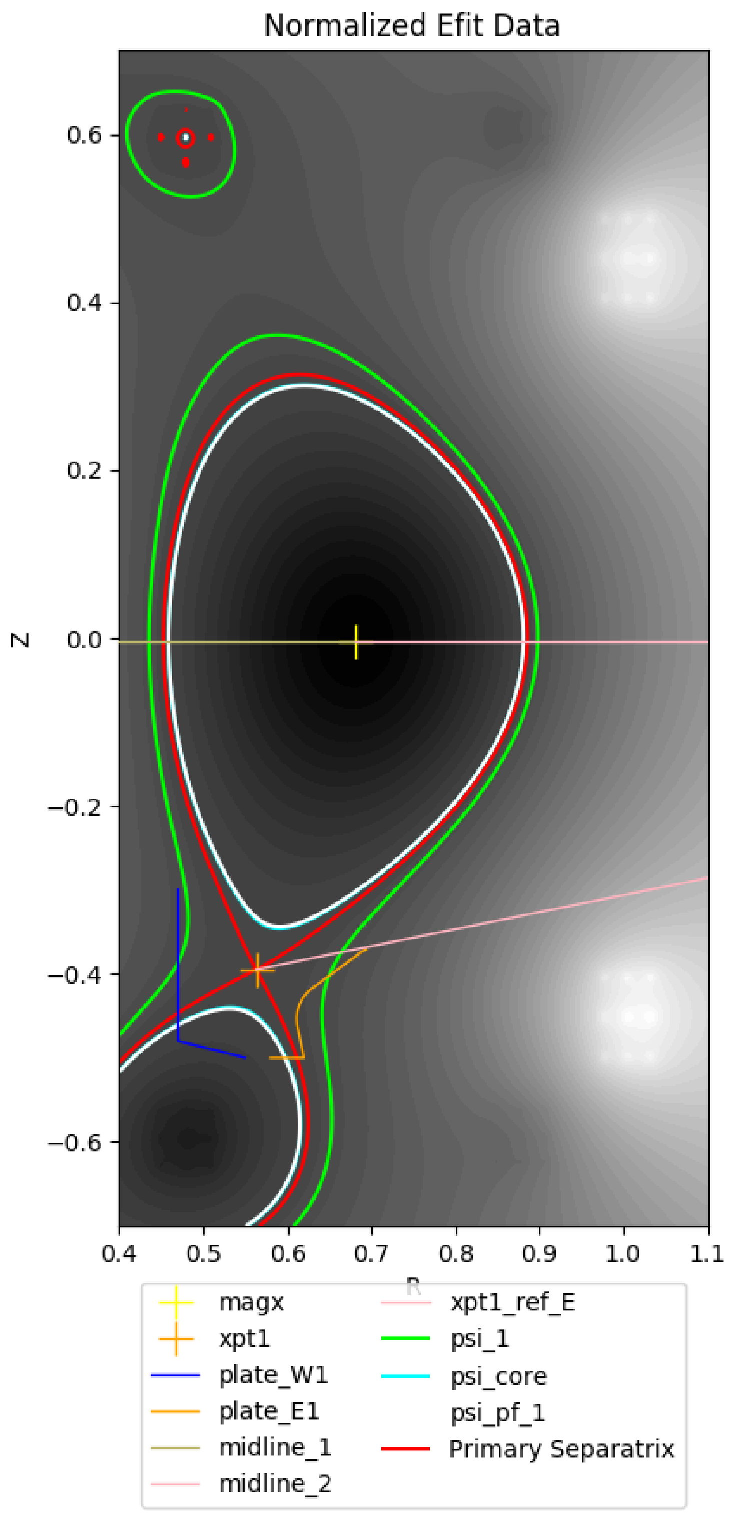
Here we will detail said cases, and also dig deeper into INGRID's capabilities for generating a grid. This tutorial will:

- Detail when adjustment to line-tracing algorithm is required by user
- Illustrate how to make adjustments to a generated `Patch` map
- Illustrate how to apply poloidal/radial transformations for non-uniform grid spacing
- Demonstrate how to reduce cell-shearing (increase orthoganality) of a grid via `distortion_correction`

2.5.1 Loading our example

The parameter file `cmod_param.yml` we will use in this tutorial is located in `example_files/SNL`.

Loading the parameter file in the GUI and viewing the data should show the following.

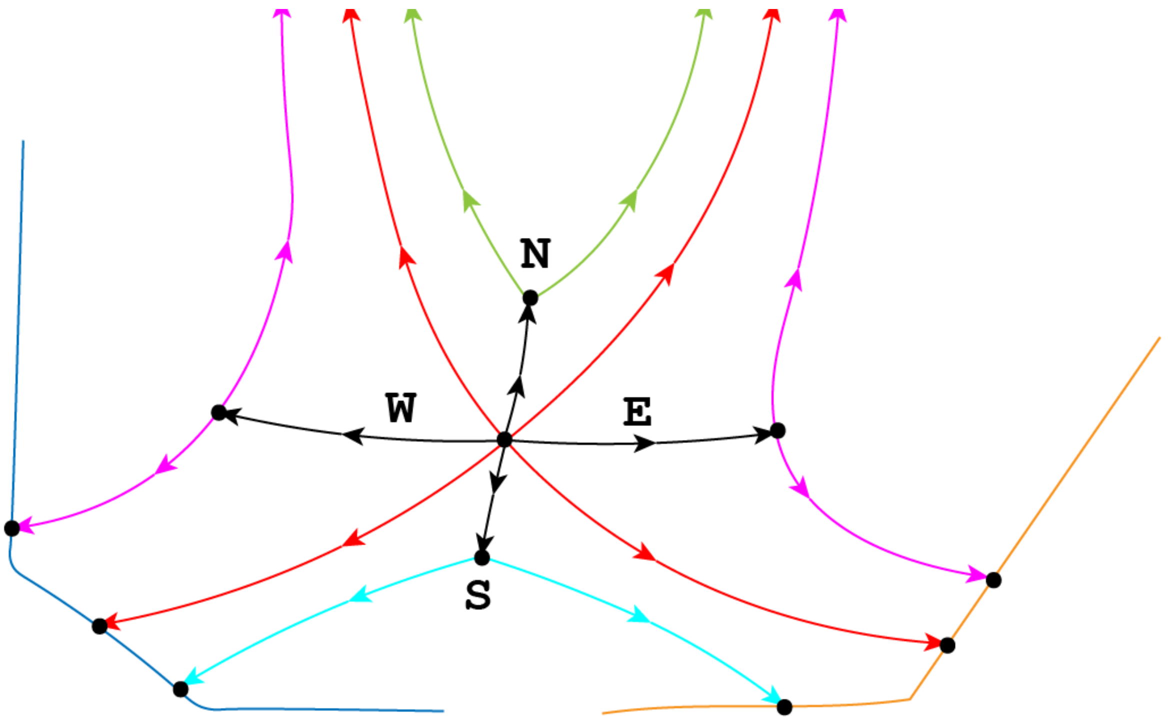

Normalized Efit Data
Z
R
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
0.4
0.5
0.6
0.7
0.8
0.9
1.0
1.1
magx
xpt1
plate_W1
plate_E1
midline_1
midline_2
xpt1_ref_E
psi_1
psi_core
psi_pf_1
Primary Separatrix

Immediately we see that there is a line segment originating from the primary x-point and extending to the EFIT domain boundary.

This is an indicator that INGRID will be overriding the default line tracing behavior from the primary x-point. As for why and how we activate this capability will be detailed in the next section.

2.5.2 Standard SNL primary x-point line tracing pattern

INGRID utilizes specific line tracing procedures for each supported topology. Below is a cartoon of line tracing directions from the primary x-point.

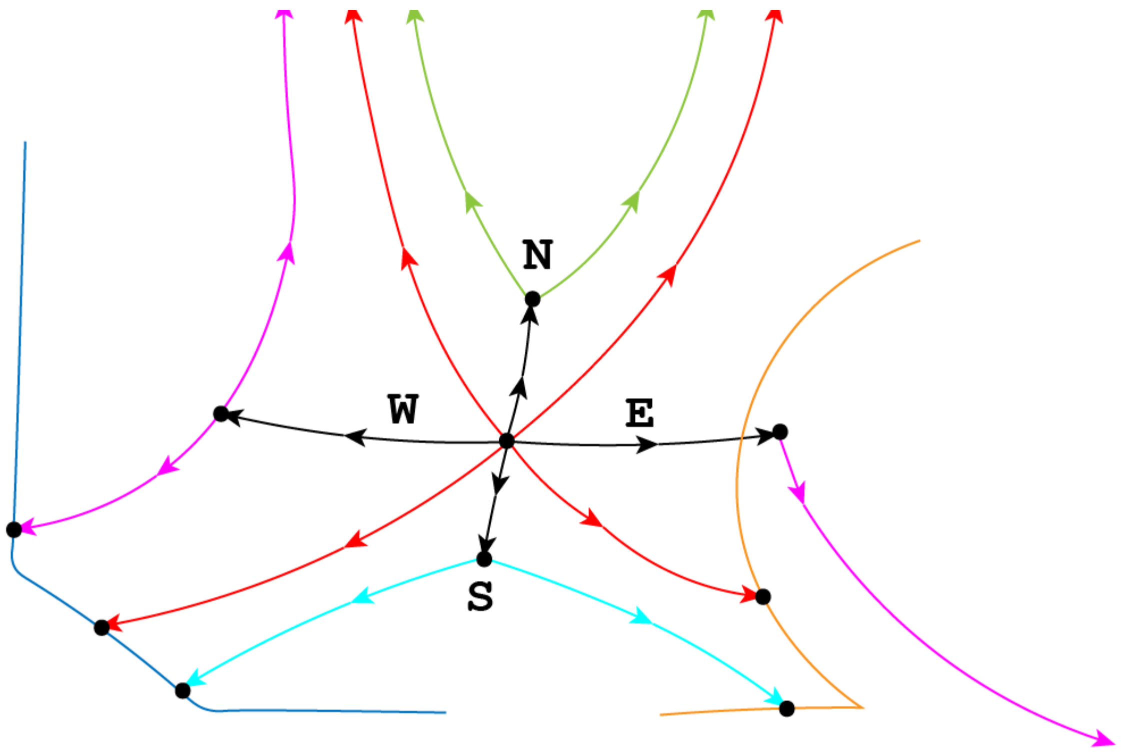


Tracing in direction **N, S, E, W** are orthogonal to flux surfaces.

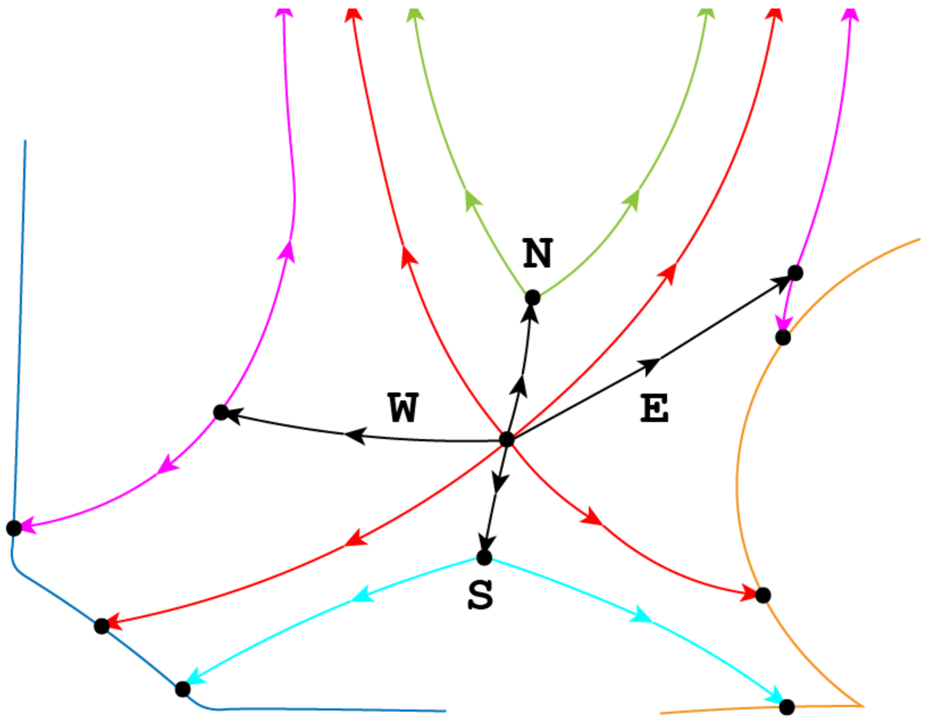
Note that line tracing from the **W** and **E** directions terminate upon intersection with the psi-max surface. Upon intersection with the psi-max surface, line tracing continues along the poloidal line and searches for intersection with a target plate.

In this particular example case we are exploring, intersection with the psi-max surface occurs past the target plate, thus causing line tracing to fail.

We see this in the cartoon below when modifying the target plate geometry.



Although this can indeed be remedied by modifying the target plate geometry or adjusting psi-max surfaces, INGRID allows the user to override the default orthogonal line tracing so that line tracing can continue without error. This remedy is illustrated below.



2.5.3 Overriding SNL primary x-point line tracing pattern

We can override the default orthogonal line tracing for both **E** and **W** directions with the entries `use_xpt1_E` and `use_xpt1_W` that reside within `patch_generation`. This can be seen below.

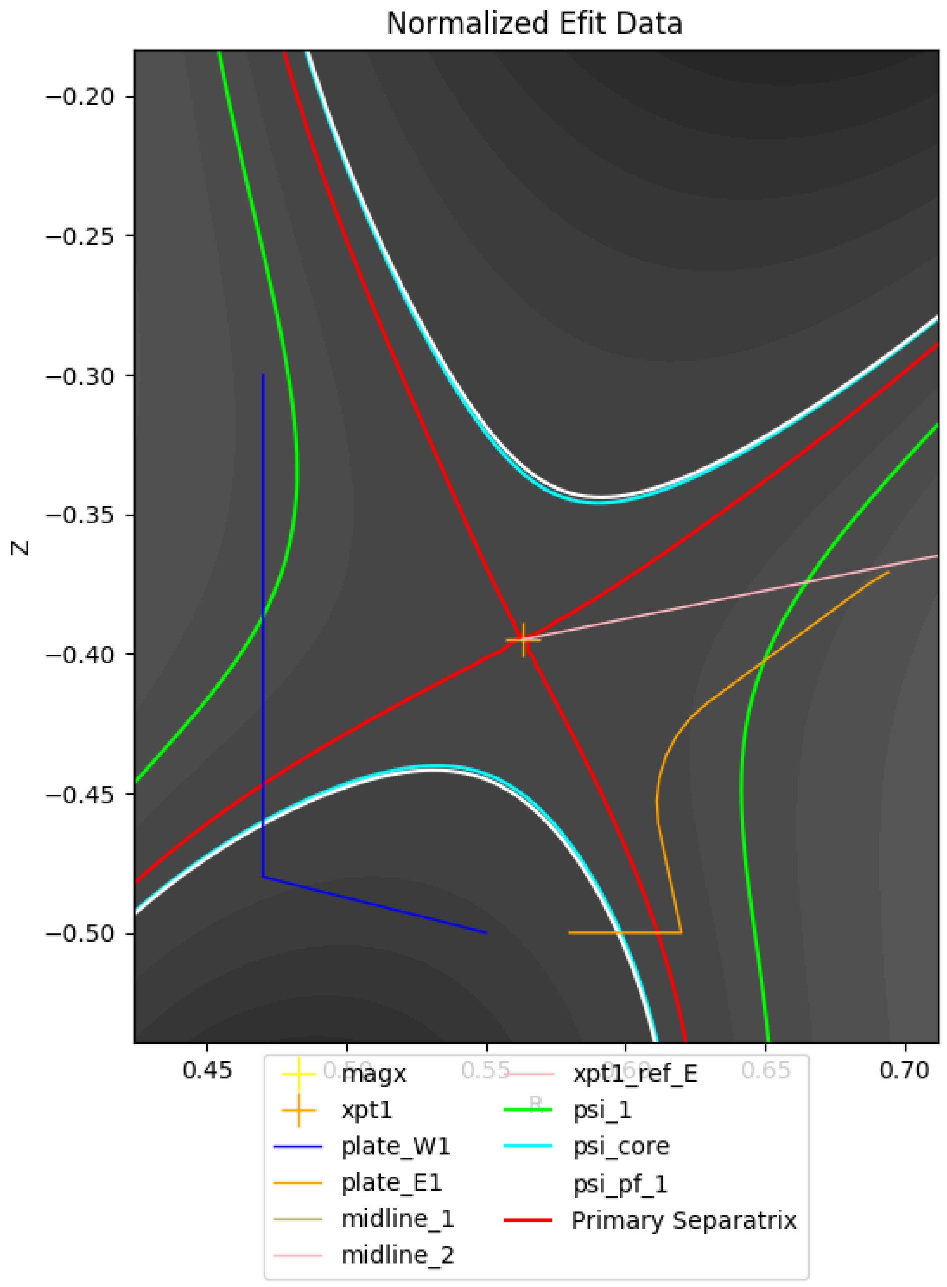
```
grid_settings:
    #...
    # other settings
    #...
    patch_generation:
        use_xpt1_E: true
        use_xpt1_W: false
```

Upon activating either entry and reloading the view into the parameter file, we will see line segment that extends from the primary x-point. This segment is a marker indicating the new line tracing direction.

By default, no rotation is applied to the line tracing direction. We can adjust the direction with the entries `xpt1_E_tilt` and `xpt1_W_tilt`. We see this below.

```
grid_settings:
    #...
    # other settings
    #...
    patch_generation:
        use_xpt1_E: true
        use_xpt1_W: false
        xpt1_E_tilt: 0.2  # radian value for rotation
        xpt1_W_tilt: -0.8  # radian value for rotation
```

The user now has the tools to remedy the above situation. We can see in this case that `xpt1_E_tilt: 0.2` provides enough clearance such that intersection with the target plate will occur.


Normalized Efit Data
Z
R
−0.20
−0.25
−0.30
−0.35
−0.40
−0.45
−0.50
0.45
0.50
0.55
0.60
0.65
0.70
magx
xpt1
plate_W1
plate_E1
midline_1
midline_2
xpt1_ref_E
psi_1
psi_core
psi_pf_1
Primary Separatrix

Tip: When it is not immediately obvious from the loaded EFIT data that orthogonal line tracing will intersect psi-max past the target plate, the user can change the visualization of the INGRID data from

filled contours to unfilled. We do this by changing `view_mode: filled` to `view_mode: lines`. We can control the number of contour lines plotted by modifying the `nlevs` entry as well. This can help with visually imagining where orthogonal line tracing will terminate.

2.5.4 Other settings for Patch map modification

Overriding orthogonal line tracing from the primary x-point is just one modification that can be made to influence a final Patch map.

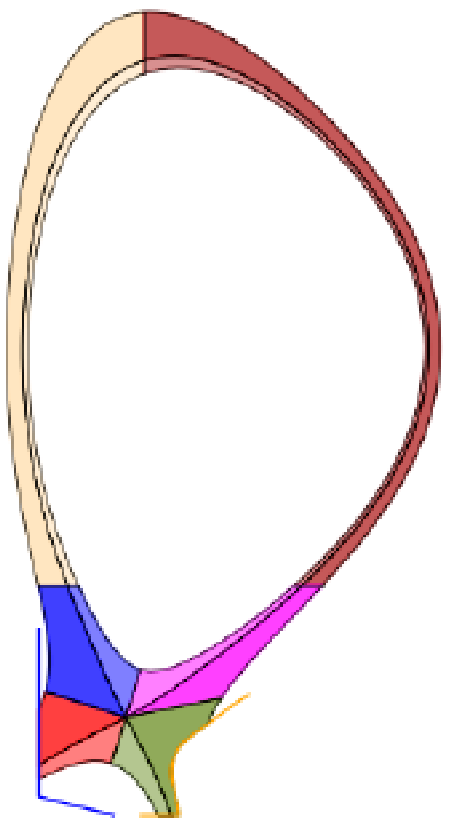
SNL line tracing for certain patches in the core will define boundaries based off intersection with the **horizontal** and **vertical** lines that intersect the magnetic axis (midplane).

One such modification is applying an RZ translation to the magnetic-axis coordinate used to generate said Patch boundaries.

This can be controlled in the parameter file by editing entries `rmagx_shift` and `zmagx_shift` under `patch_generation` in `grid_settings`.

```
grid_settings:
    patch_generation:
        rmagx_shift: 0.0  # Translate R coordinate
        zmagx_shift: 0.0  # Translate Z coordinate
```

Saving the parameter file and reloading the view into the data will reflect the changes. The Patch map generated with the translations above can be seen below.

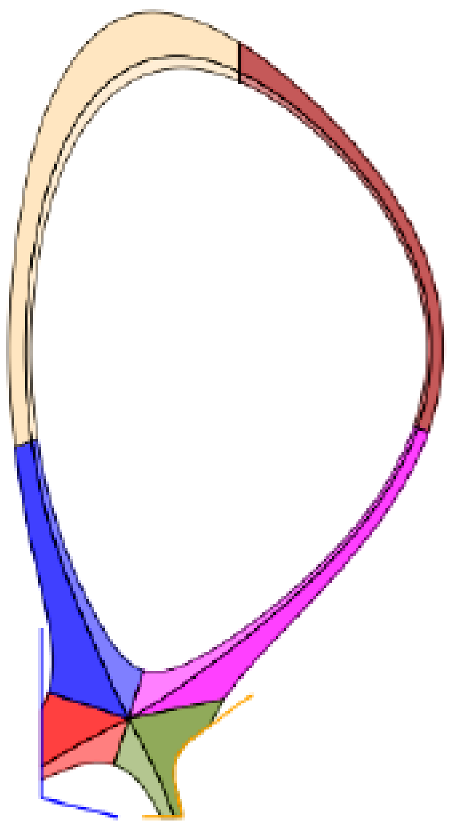


In a similar manner to adjusting the angle of line tracing in the E and W directions from the primary x-point, we can adjust the line segments extending from the magnetic-axis. These line segments define the east faces of patches B1 and B2, as well as the west faces of patches E1 and E2.

The tilt of the inner-midplane and outer-midplane can be controlled with entries magx_tilt_1 and magx_tilt_2 respectively. These entries are contained within patch_generation in grid_settings.

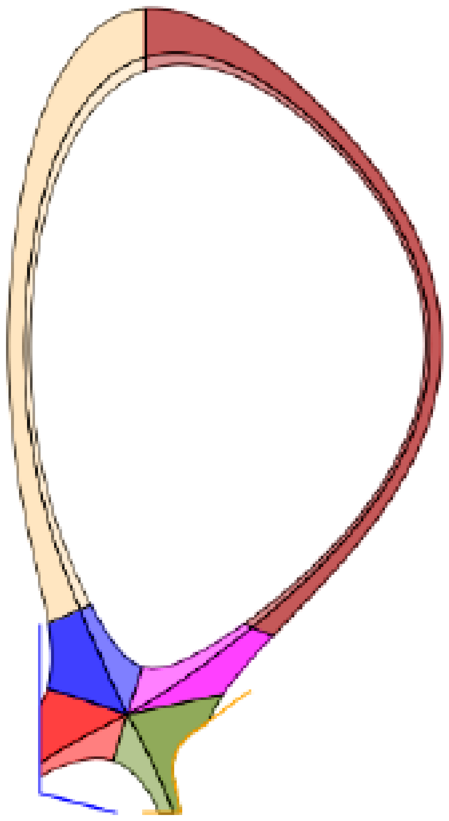
```
grid_settings:
    patch_generation:
        magx_tilt_1: 0.0  # inner-midplane rotation (in radians)
        magx_tilt_2: 0.0  # outer-midplane rotation (in radians)
```

Saving the parameter file and reloading the view into the data will reflect the changes. The Patch map generated with the only the tilt values entered above can be seen below.



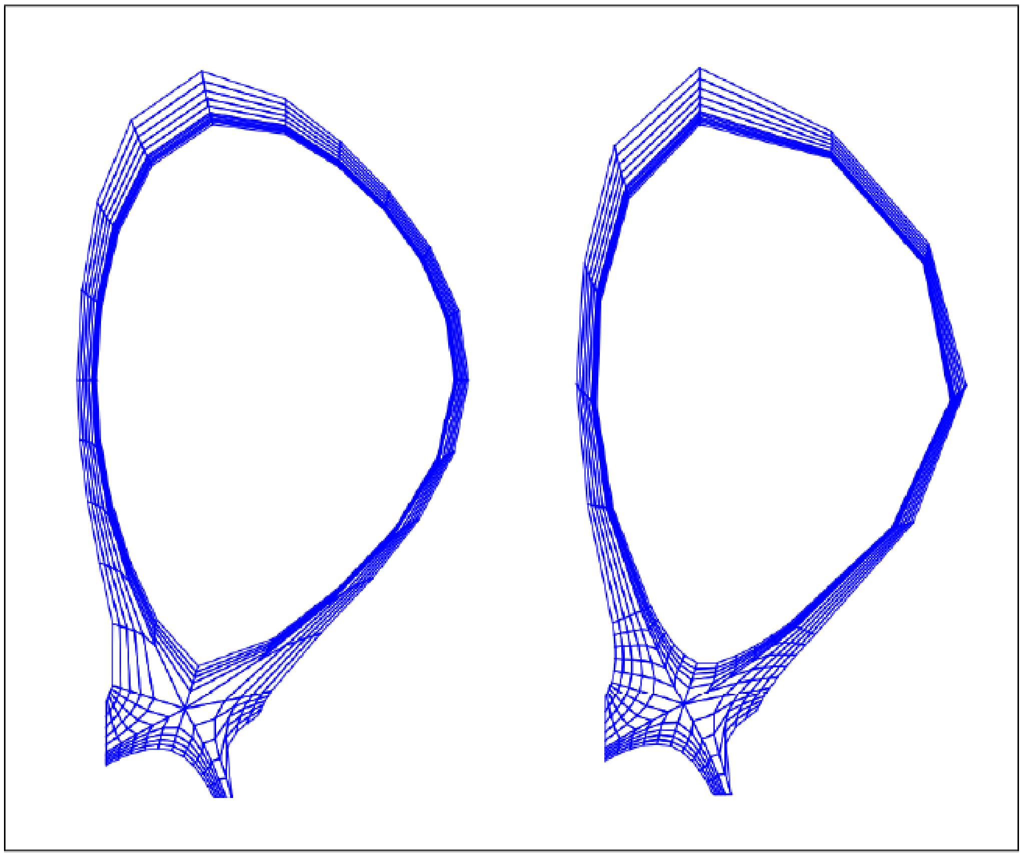
Note: Midplane tilt entries are in radians and follow the standard counter-clockwise rotation direction.

Combining both together yields the following Patch map.



Tip: Applying these Patch modifications appropriately can allow one to increase cell density near primary x-point without modifying np/nr values

On the left is the grid with no Patch map modifications for reference.

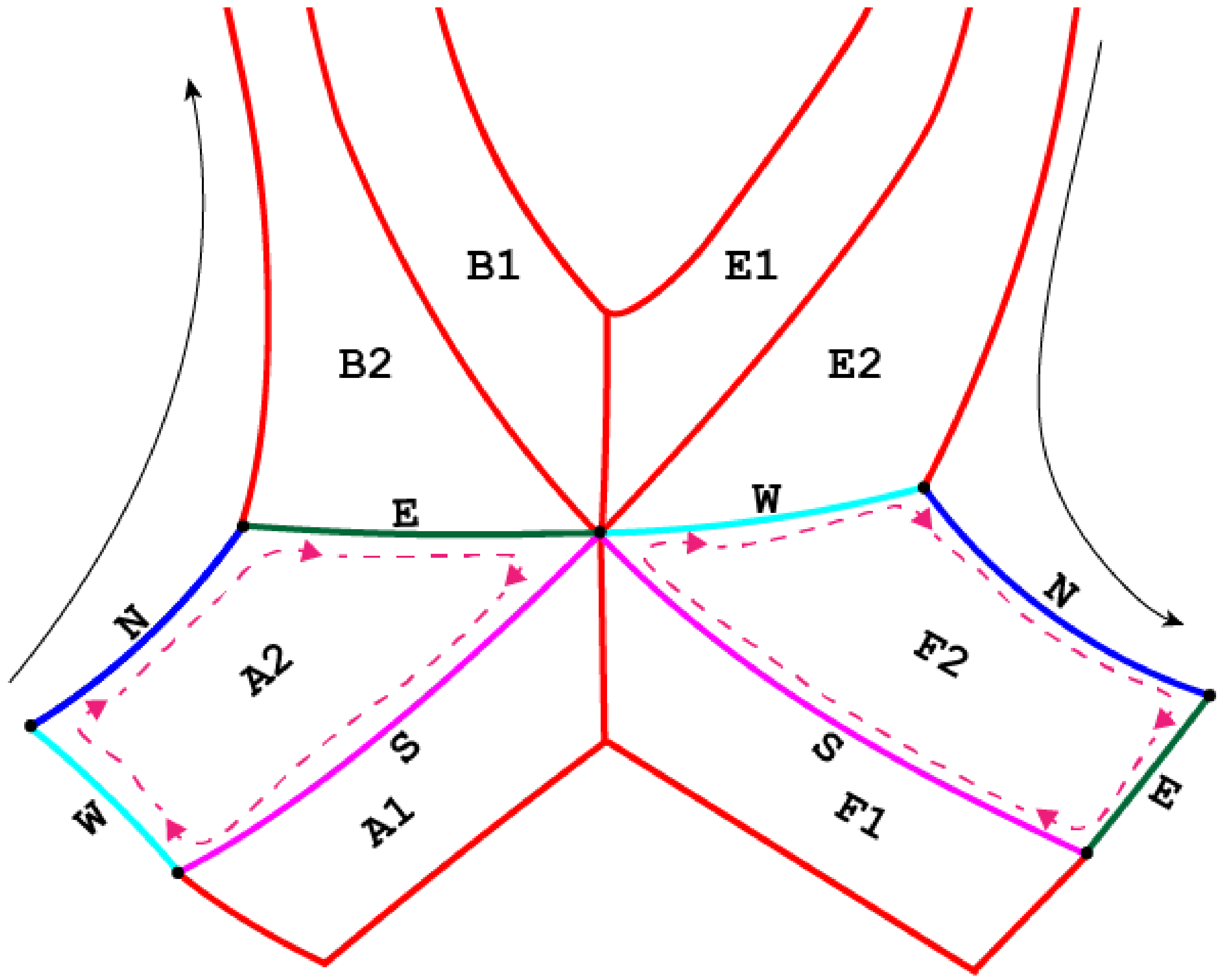


2.5.5 Background knowledge for poloidal and radial grid transformations

INGRID allows the user to provided poloidal and radial grid distribution functions for generating non-uniform grids.

Before detailing how to invoke these features, some background on the Patch object itself.

Each Patch boundary is defined by 4 lines that we refer to as **N**, **E**, **S**, **W**. This allows for us to maintain a clockwise orientation on the boundary of a Patch. Below is a cartoon illustrating the idea.



The **N** boundary (seen in dark blue) for Patch A2 begins at the max-psi strike-point on the target plate west of the primary x-point (inner target plate for SNL case), and terminates at the B2 interface. The **S** boundary (seen in magenta) for Patch A2 is oriented in the opposite direction and terminates upon intersection with the target plate.

The **E** and **W** boundaries (seen in dark green and cyan, respectively) are defined in the radial direction relative to the **N** and **S** boundaries.

Note that this convention holds throughout the entire Patch map. We can see this by noticing that upon reaching Patch `F2`, the **E** boundary is now defined by a portion of the (LSN outer) target plate.

Note: INGRID chooses to parameterize the **N face in length with parameter $s \in [0, 1]$ for poloidal distribution functions. Similarly, INGRID chooses to parameterize the **W** face in increasing psi with parameter $s \in [0, 1]$ for radial distribution functions. **

Now we discuss how the user specifies poloidal and radial grid transformations within the parameter file.

INGRID parses a string from the user in the form ``x, f(x)`` where :math:`x` indicates the dependent variable and :math:`f(x)` is mathematical expression representing the distribution. Within the parameter file, we have seen the string `x, x` utilized for entries `radial_f_default` and `poloidal_f_default`. INGRID interprets this as applying a uniform distribution of vertices for defining the grid (consistent with what we have seen).

Warning: Due to the parameterization $s \in [0,1]$, defining $f(x)$ such that $f : [0,1] \to [0,1]$ is important. Apply appropriate normalization operations when utilizing non-trivial functions (see example below).

INGRID utilizes SymPy for generating a function from the user provided string. Standard Python arithmetic operations are supported (+, -, *, /, **, ...), as well as common mathematical functions such as `exp` and `log`.

2.5.6 Applying poloidal and radial grid transformations

In general, we adopt a notation similar to specifying np/nr cells. Below is a snippet of a YAML file with default poloidal and radial transformation values.

```
grid_settings:
    grid_generation:

        # ...
        # Other grid_generation settings
        # ...

        poloidal_f_default: x, x  # Global uniform poloidal
        radial_f_default: x, x    # Global uniform radial
```

Much like `np_default` and `nr_default`, entries `poloidal_f_default` and `radial_f_default` apply to poloidal "columns" and radial "rows" in index space, respectively. The `default` appended to `poloidal_f_` and `radial_f_` tells INGRID to apply the corresponding transformation globally.

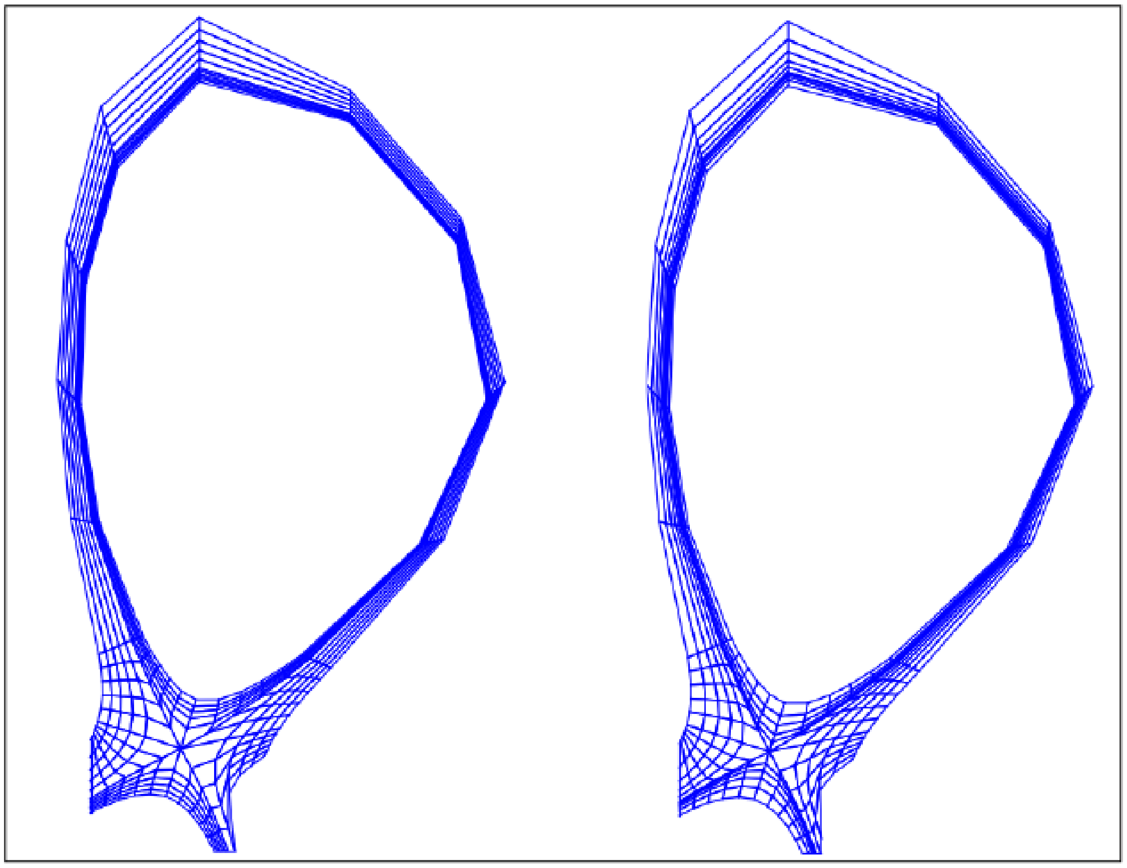
Poloidal transformations can be specified with the same convention as specifying poloidal cells (`poloidal_f_A`, `poloidal_f_B`, ..., `poloidal_f_F`).

Radial transformations follow the same convention (`radial_f_1`, `radial_f_2`), but also have an additional `radial_f_3` specifically for the inner-core region.

The following applies an exponential-like distribution (between 0 and 1) for the SOL, PF, and CORE. These transformations will generate grid cells that hug the primary separatrix slightly more than usual.

```
grid_settings:
    grid_generation:

        # ...
        # Other grid_generation settings
        # ...

        poloidal_f_default: x, x  # Global uniform poloidal
        radial_f_default: x, x    # Global uniform radial
        radial_f_1: x, 1-(1-exp(-(1-x)/0.4))/(1-exp(-1/0.4))
        radial_f_2: x, (1-exp(-(x)/0.8))/(1-exp(-1/0.8))
        radial_f_2: x, (1-exp(-(x)/0.8))/(1-exp(-1/0.8))
```

The resulting grid with transformations can be seen on the left, and the original grid with no transformations can be seen on the right.



2.5.7 Reducing cell shearing via `distortion_correction`

INGRID does not enforce an orthogonality condition when generating a grid. INGRID allows the user to impose angle constraints on cells within a generated grid in order to increase orthogonality. We do this via the `distortion_correction` feature.

Below is an example of cell shearing and the motivation for INGRID's `distortion_correction`.

This `distortion_correction` tool allows the user to specify angle constraints `theta_min` and `theta_max` in order to mitigate cell shearing. INGRID will shift the cell vertex by increments of 1 / `resolution` until the resultant angle is within the user constraints.

If the constraint cannot be satisfied (vertex leaves the Patch), INGRID will backtrack until the vertex is within the Patch bounds.

Below is a snippet of the parameter file format showing `distortion_correction` applied globally.

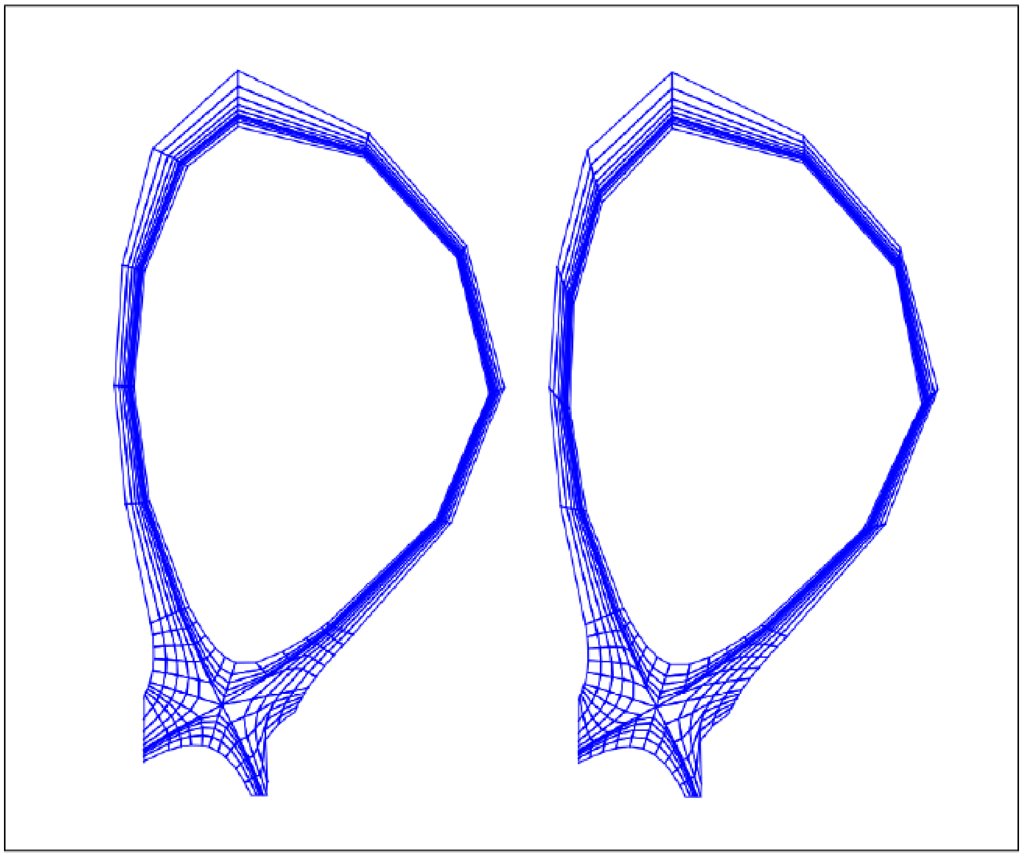
```
grid_settings:
    grid_generation:
        distortion_correction:
```

(continues on next page)

(continued from previous page)

```
        # Global settings
        all:
            active: True  # toggle distortion_correction
            resolution: 1000  # 1 / resolution step-size for shifting vertex
            theta_max: 120.0  # angle constraint
            theta_min: 80.0   # angle constraint

        # Patch specific settings can be provided in addition to global settings
        # (similar to how we specify np/nr for Patches on top of default np/nr)

        # Example: Specify distortion_correction for Patch A1

        # A1:  # <-- (Patch name here can be changed)
        #    active: false  # toggle distortion_correction
        #    resolution: 1000  # 1 / resolution step-size for shifting vertex
        #    theta_max: 120.0  # angle constraint
        #    theta_min: 80.0   # angle constraint

    np_default: 5
    nr_default: 5
    poloidal_f_default: x, x
    radial_f_default: x, x
```

Below is a side-by-side comparison of `distortion_correction` toggled on and off respectively.



We can see that the radial lines are indeed more orthogonal to the poloidal contours. Although only a mild effect in this case, we have seen that `distortion_correction` can significantly reduce shearing and generate tidier grids in others (example SF cases seen below).

This feature (in addition to those detailed above) is just another tool at a user's disposal that need not be utilized in every case.

2.5.8 Adjusting guard cell size

Guard cell size for a generated grid can be specified within the parameter file by editing entry `guard_cell_eps` within `grid_settings`. That is:

```
# ---------------------------------------------------
# General grid settings
# ---------------------------------------------------
grid_settings:
    # ...
    # ... Other grid_settings entries
```

(continues on next page)

(continued from previous page)

```
    # ...
    guard_cell_eps: 0.00001  # Size of guard cells.
```

Note: Specification of guard cell size must be done prior to initiating grid generation (clicking `Create Grid`).

2.5.9 Summary

In this tutorial, we encountered a situation where parameter file modification is required for INGRID to successfully generate a Patch map. This was resolved by modifying the line tracing procedure in order to accomodate the provided geometry.

We also saw how the modification of the line tracing procedure falls into the over-arching category of INGRID tools that allow the user to customize a Patch map.

Finally, we dove deeper into grid customization capabilities such as applying `distortion_correction`, poloidal and radial transformations, and specifying guard cell size.

2.6 Example: Two x-points in domain (SF75 example)

Note: This tutorial assumes the reader has already read both the introductory SNL tutorial and the further exploration SNL tutorial.

Here we will demonstrate how to generate a grid when there are two x-points in the domain. In particular, we will generate a grid for a snowflake-75 (SF75) configuration.

Because INGRID internally handles the identification and classification of magnetic topology, the general steps detailed here for generating a grid apply to **all** the other configurations with two x-points in the domain. The user will only need to refer to provided diagrams of a configuration's Patch map, x-point NSEW directions, and psi labels for a specific configuration.

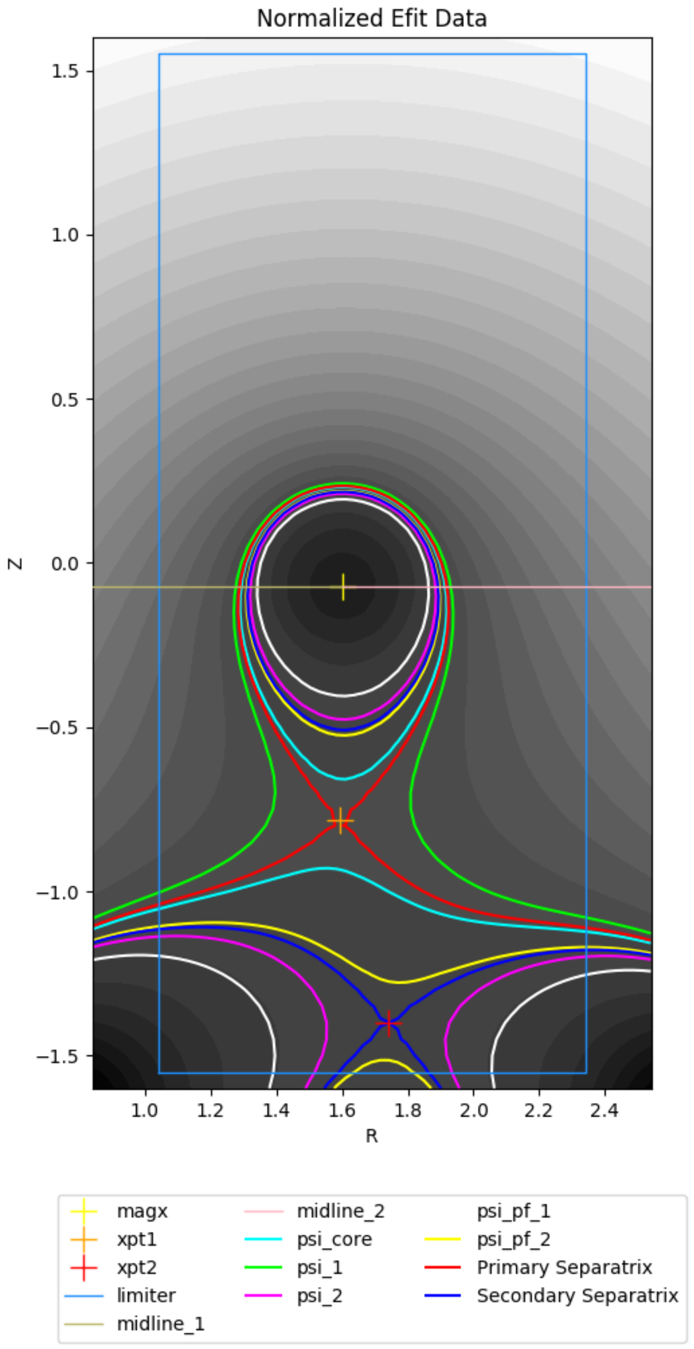
2.6.1 Loading our example

The parameter file `SF75.yml` we will use in this tutorial is located in `example_files/SF75`.

A key difference inside the parameter file is that now the entry `num_xpt` has an associated value of `2`. This **activates** INGRID's topology classification when two x-points are present in the domain. With this, INGRID now expects more parameter file entries from the user. These include:

- Approximate R coordinate of secondary x-point `rxpt2`
- Approximate Z coordinate of secondary x-point `zxpt2`
- Additional psi entries `psi_2` and `psi_pf_2`
- Limiter data (or usage of EFIT domain bounds to act as a psuedo-limiter)
- Two additional target plates (when `strike_pt_loc` has an associated value of `target plates`)

Loading the parameter file in the GUI and viewing the data should show the following.

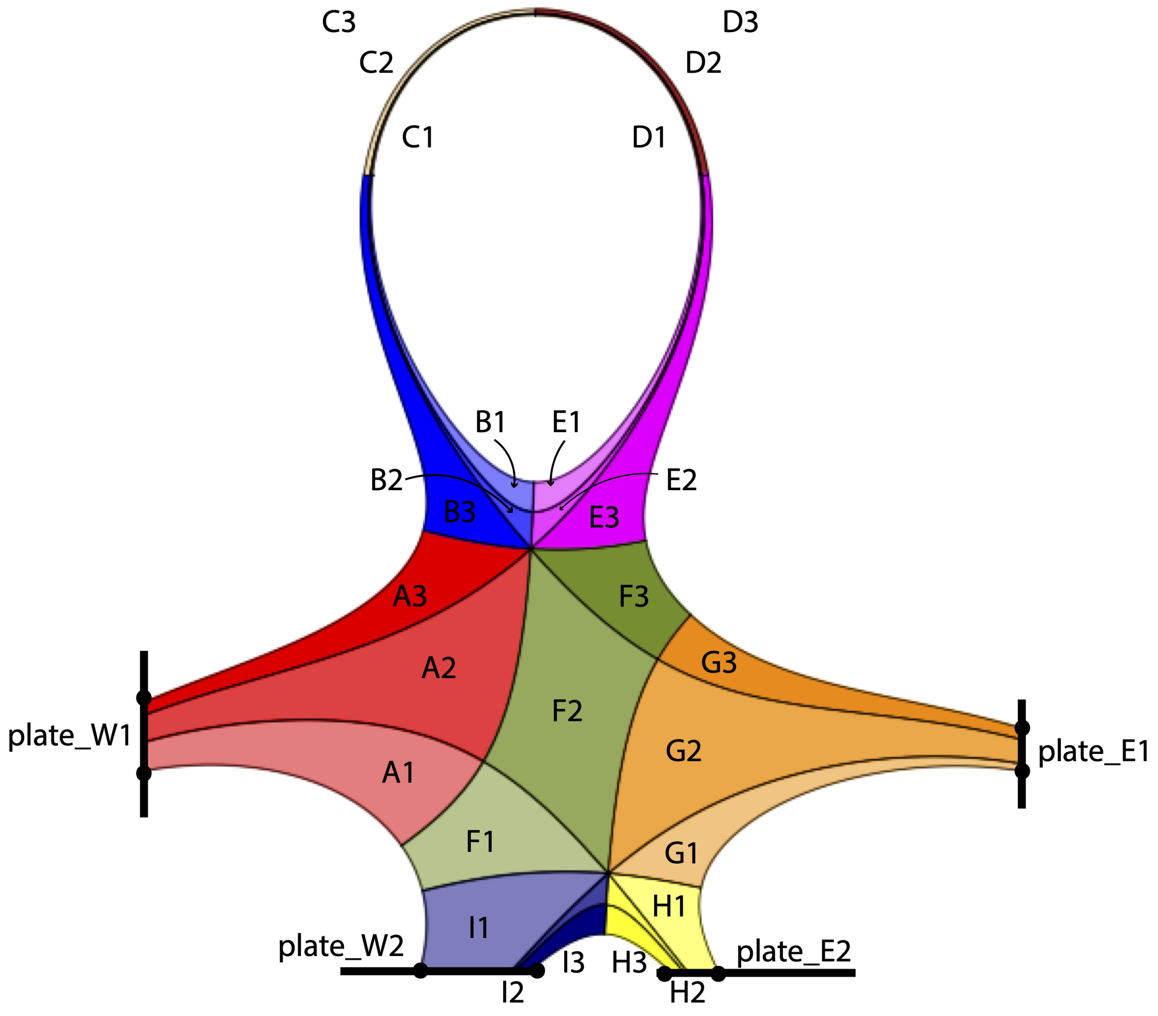


We can immediately see the presence of a secondary x-point, it's corresponding separatrix, a rectangular (psuedo) limiter and additionaly psi lines that we will use to generate a Patch map.

Before generating a Patch map, we discuss the SF75 layout.

2.6.2 The SF75 line-tracing pattern, x-point directions, and psi labels

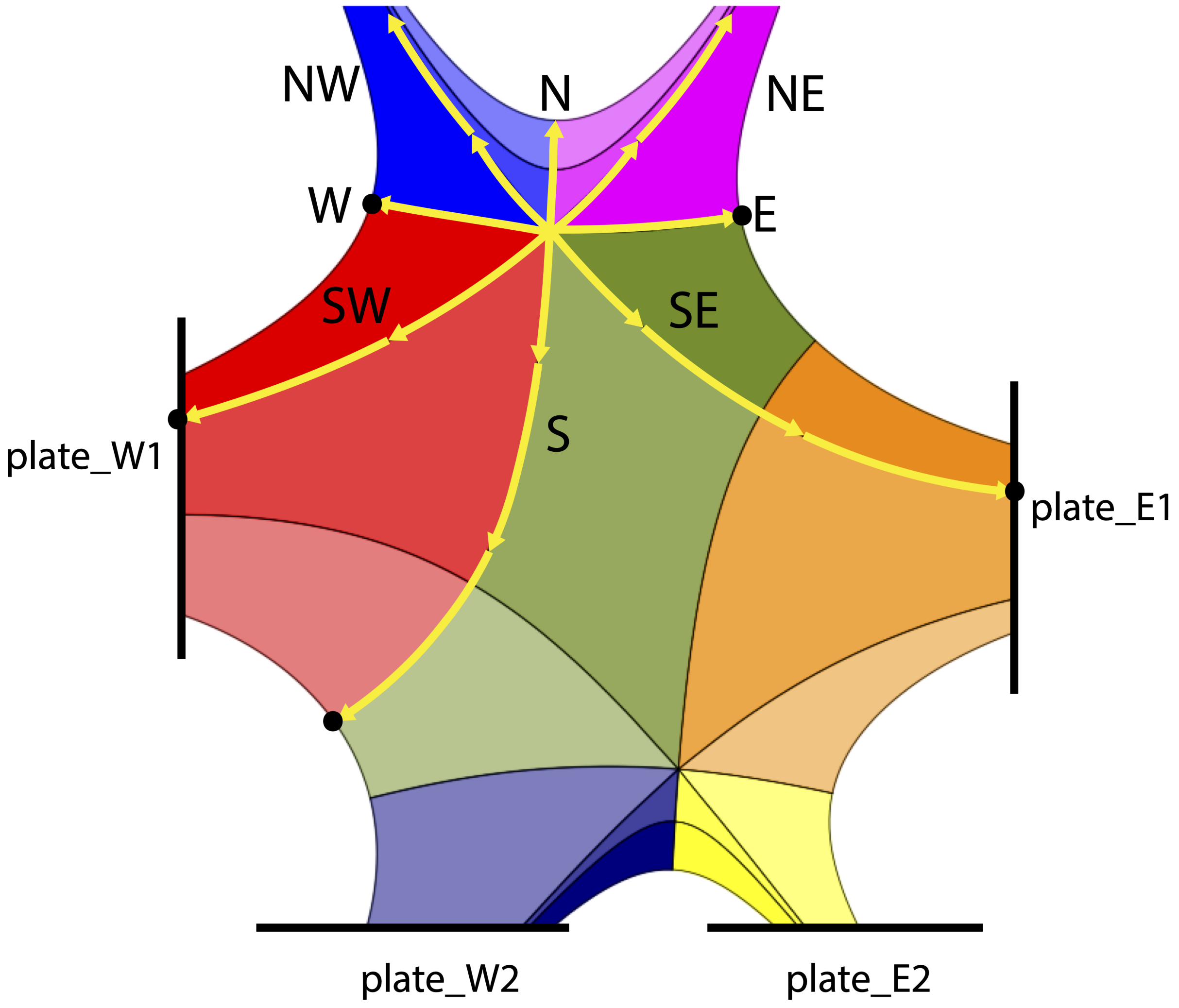
The figure below shows a Patch map for general SF75 configuration.

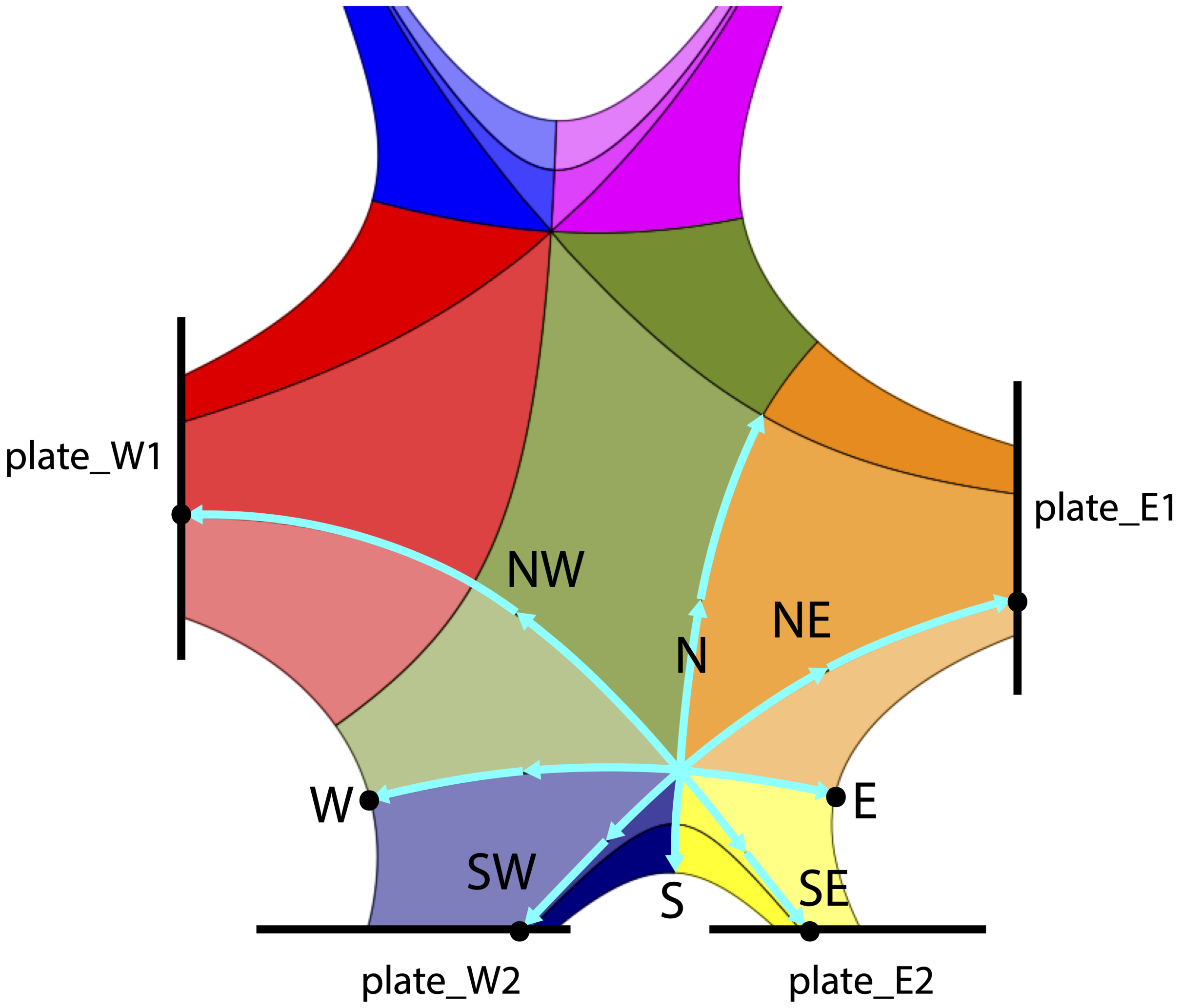


The user should reference the above figure when prepping the parameter file for generating a Patch map by verifying target plates are in valid locations relative to x-points and psi-boundaries.

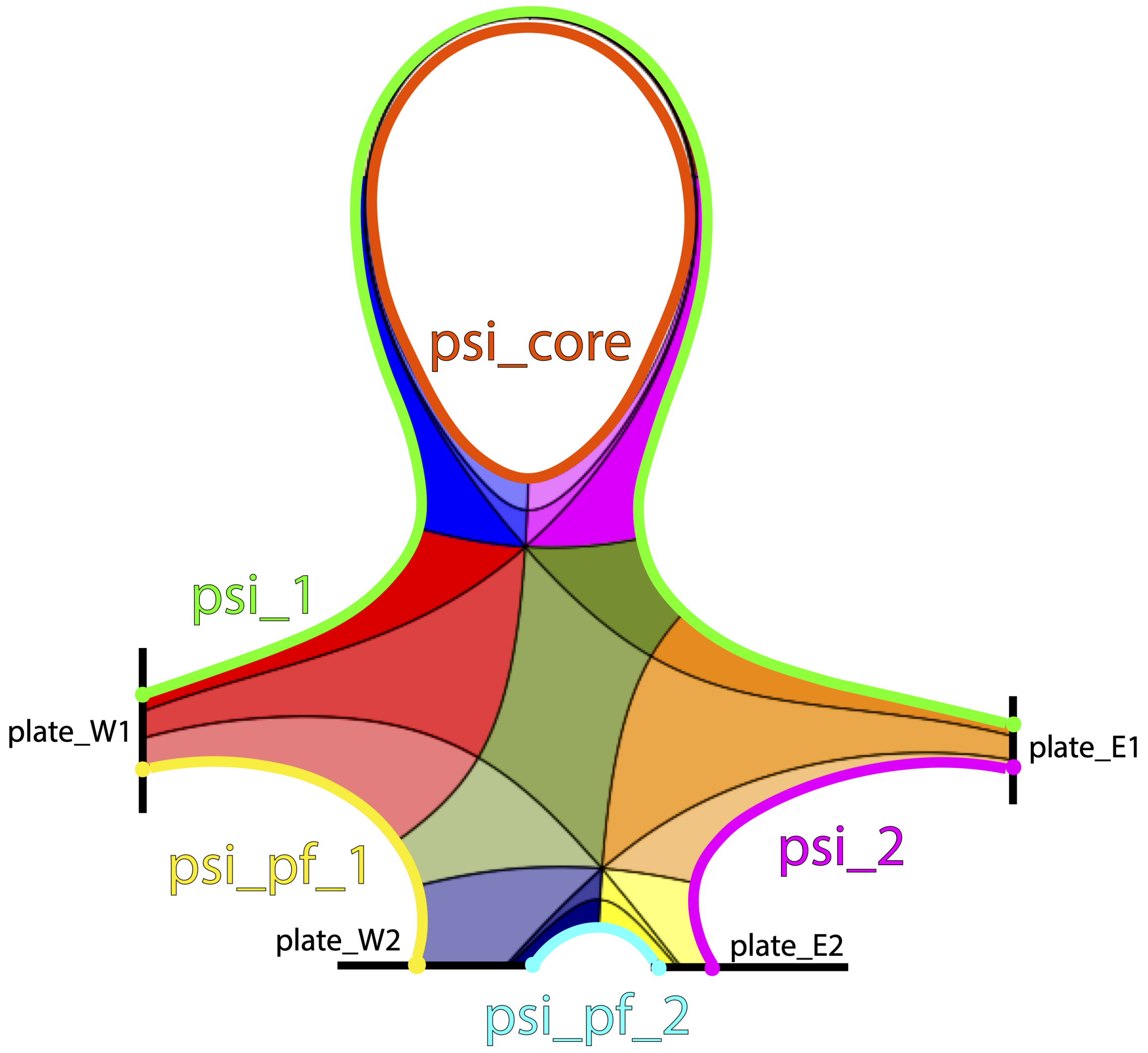
First, note the target plate naming convention `plate_W1`, `plate_E1`, `plate_W2`, and `plate_E2`. Again, by removing notions of "inner" and "outer" plates in the `SNL` case, we see that we now have a general naming convention that can be applied to all configurations with two x-points in the domain.

This naming convention is defined by the NSEW directions for both the primary x-point (`xpt1`) and secondary x-point (`xpt2`). Below we can see two figures that illustrate the NSEW directions for `xpt1` and `xpt2`, respectively.


NW
N
NE
W
E
SW
SE
S
plate_W1
plate_E1
plate_W2
plate_E2



Finally, we illustrate the psi-labels associated with the psi-boundaries of the SF75 configuration.



The above shows we now have parameter file entries `psi_2` and `psi_pf_2` to assign values to.

We will see in other configurations (e.g. `UDN`, `SF45`, etc) that the location of the psi labels **will** vary. Because the location of psi boundaries vary with each configuration, the user must carefully note which psi boundaries they are dealing with.

In general, `psi_1` will correspond to psi-max, `psi_core` remains set to the core psi value, `psi_pf_1` will correspond to the psi-surface we intersect by tracing `S` of `xpt1`, and `psi_pf_2` will correspond to the psi-surface we intersect by tracing `S` of `xpt2`. Typically, the `psi_2` psi-boundary and strike-point locations for the psi-entries listed above often vary.

2.6.3 Activating the limiter for Patch generation

Understanding INGRID limiter usage is an essential part of generating a Patch map for cases with two x-points in the domain.

INGRID does not require the use of target plates for generating a Patch map if the user opts for using a limiter.

This is controlled within the parameter file with `strike_pt_loc` within the `patch_generation` block. We see this below.

```
grid_settings:

  # ...
  # other settings
  # ...

  patch_generation:
    # ...
    # other settings
    # ...

    # strike_pt_loc takes the values of 'limiter' or 'target_plates'
    strike_pt_loc: limiter  # generates a Patch map with a limiter rather than target␣
↪plates
```

When `strike_pt_loc` is set to a value of `limiter`, INGRID will utilize all geometry data provided in the parameter file block labeled `limiter`.

We can see the `limiter` settings in `SF75.yml` below.

```
grid_settings:

  # ...
  # other settings
  # ...

  patch_generation:
    # ...
    # other settings
    # ...

    # strike_pt_loc takes the values of 'limiter' or 'target_plates'
    strike_pt_loc: limiter  # generates a Patch map with a limiter rather than target␣
↪plates

# Specifications for using a limiter
limiter:

  file: ''  # File name of .txt file with coordinates specifying limiter geometry.

  use_efit_bounds: true  # Use the EFIT domain boundary as a limiter

  # Coordinates: [(rmin, zmin), (rmax, zmin), (rmax, zmax), (rmin, zmax), (rmin, zmin)]

  zshift: 0.0  # Shift the limiter geometry in the z-direction
```

(continues on next page)

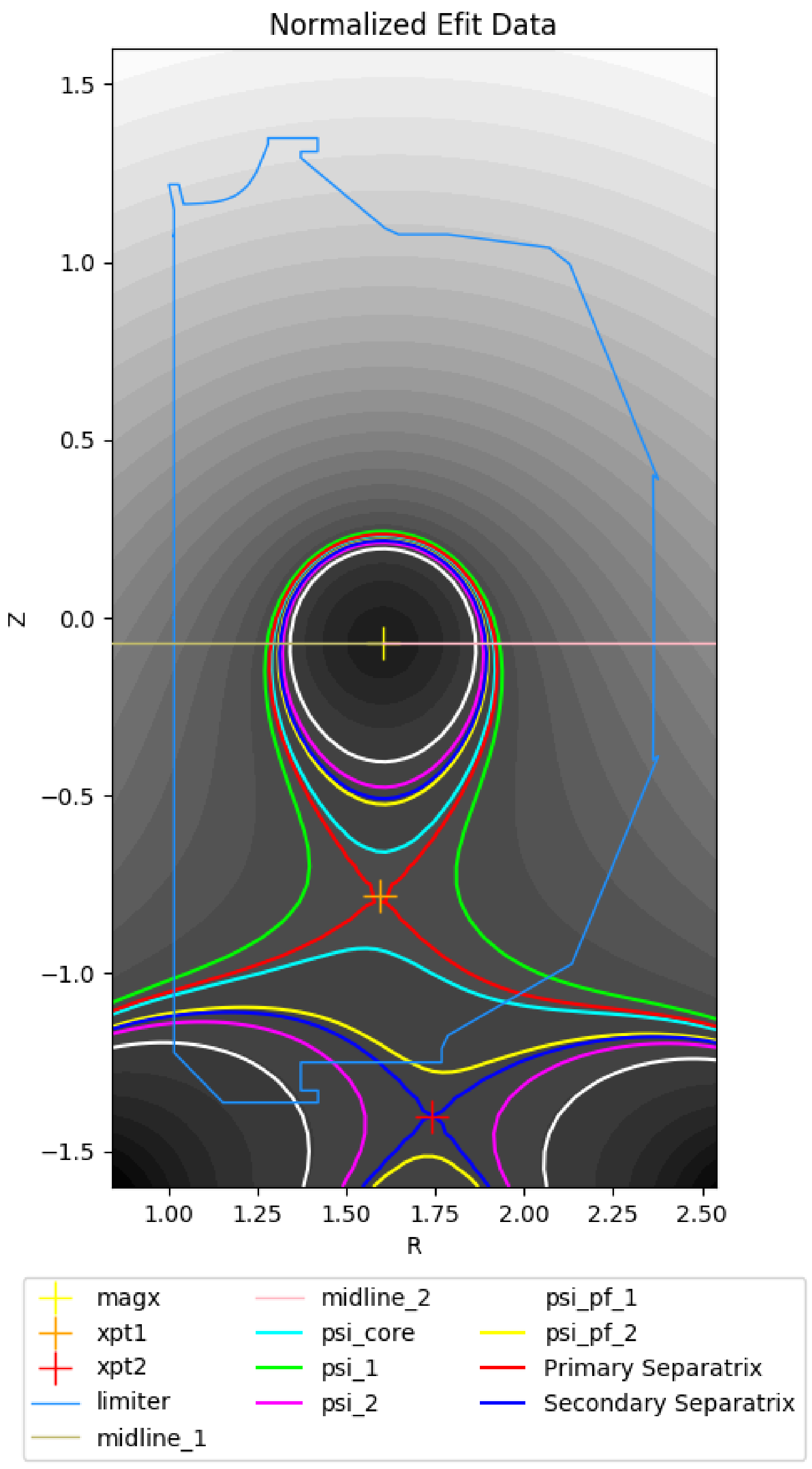
(continued from previous page)

```
  rshift: 0.0  # Shift the limiter geometry in the z-direction

  # Adjust shape of EFIT domain boundary psuedo-limiter in r coordinate
  efit_buffer_r: 0.2  # Default value: 1.0e-2

  # Coordinates: [(rmin + efit_buffer_r, zmin), (rmax - efit_buffer_r, zmin),
  #               (rmax - efit_buffer_r, zmax), (rmin + efit_buffer_r, zmax), (rmin +␣
↪efit_buffer_r, zmin)]

  # Adjust shape of EFIT domain boundary psuedo-limiter in z coordinate
  efit_buffer_z: 0.05  # Default value: 1.0e-2

  # Coordinates: [(rmin, zmin + efit_buffer_z), (rmax, zmin + efit_buffer_z),
  #               (rmax, zmax - efit_buffer_z), (rmin, zmax - efit_buffer_z), (rmin,␣
↪zmin + efit_buffer_z)]
```

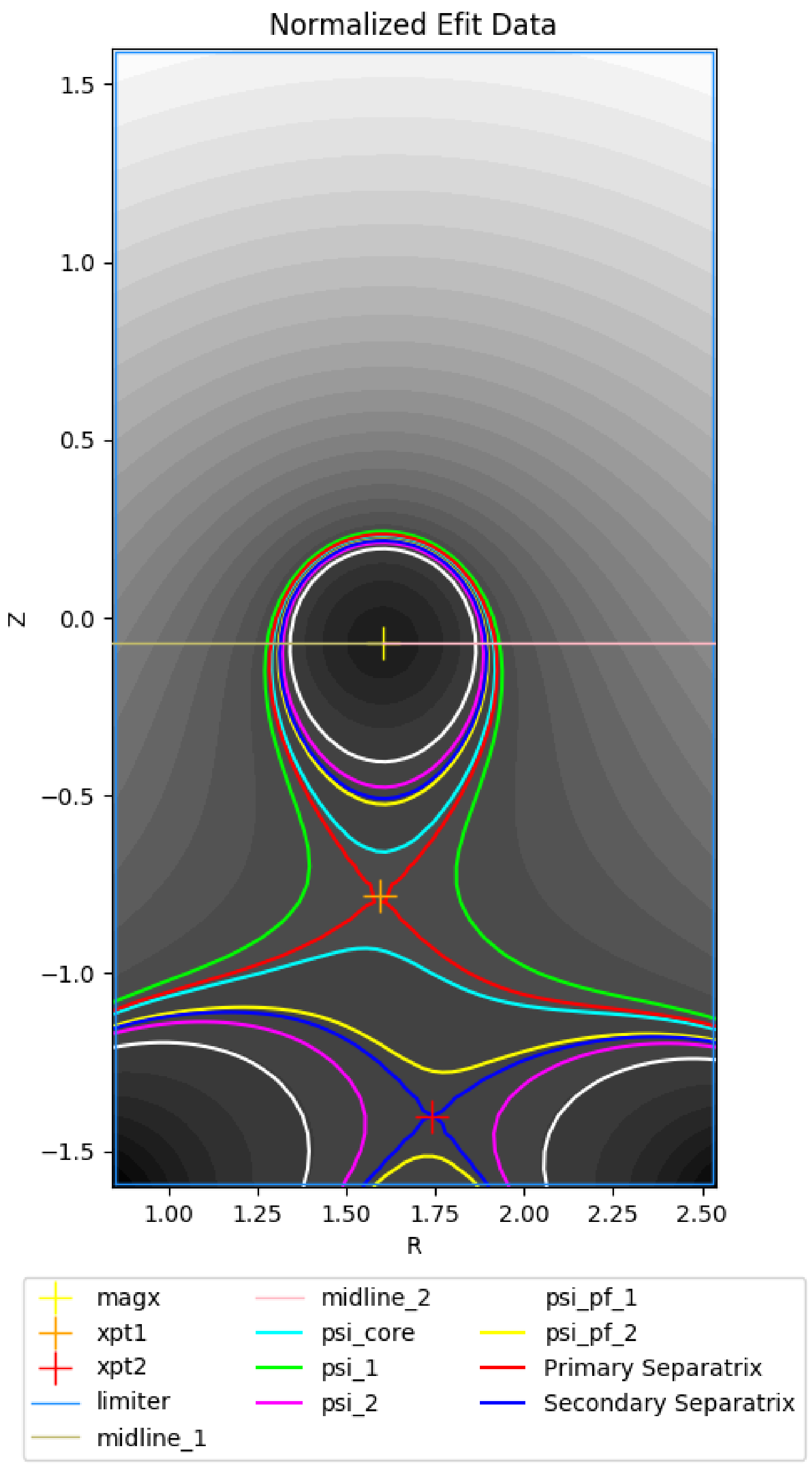
Note: INGRID will utilize the default limiter data provided within the `eqdsk` file if no file is provided and `use_efit_bounds` is set to `False`. If no limiter data is available in the `eqdsk` file, INGRID will set `use_efit_bounds` to `True`.

Below are figures illustrating possible edits to the parameter-file `limiter` block entry (we will **not** be using these values for the remainder of the tutorial).

First, setting `use_efit_bounds` to `False` (default `eqdsk` limiter data)


Normalized Efit Data
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Z
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R
magx
xpt1
xpt2
limiter
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

Next, setting `use_efit_bounds` back to `True`, but setting both `efit_buffer_r` and `efit_buffer_z` back to their **default** values of `1.0e-2`.


Normalized Efit Data
Z
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R
magx
xpt1
xpt2
limiter
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

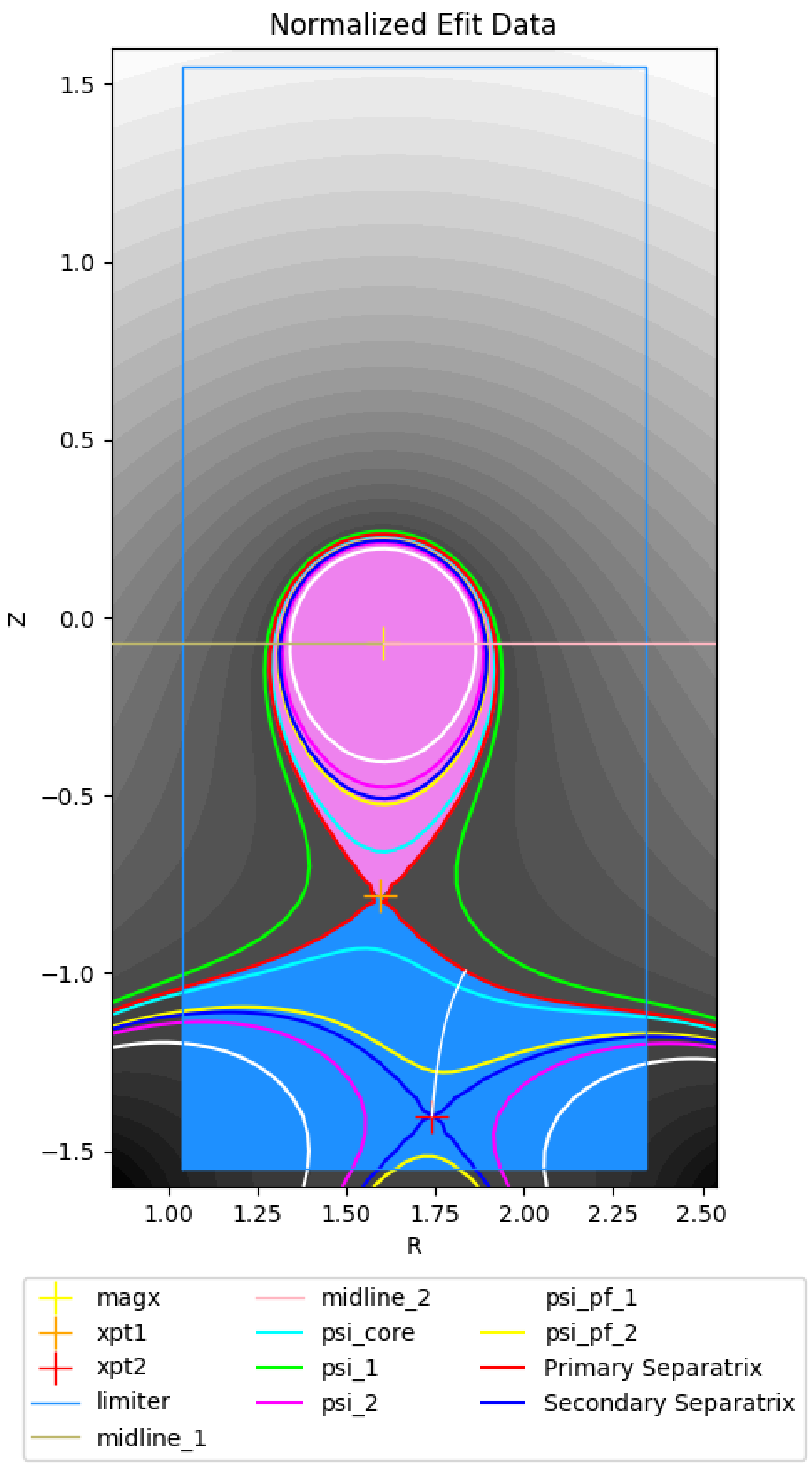
Warning: When setting `use_efit_bounds` to `True`, the user must provide a non-zero value for values `efit_buffer_r` and `efit_buffer_z`.

2.6.4 INGRID identification of configuration

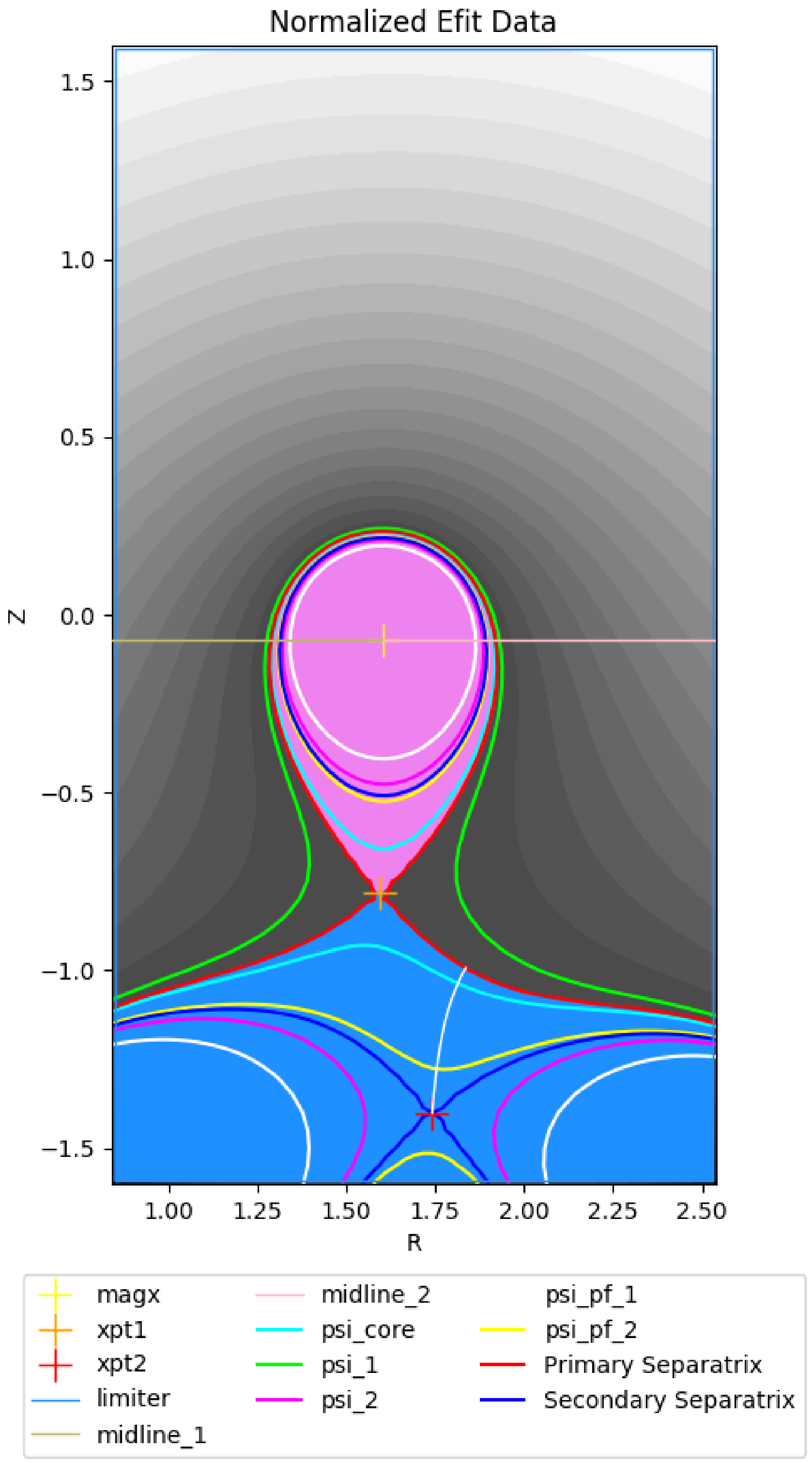
As mentioned in the previous section, understanding INGRID limiter usage is an essential part of generating a Patch map for cases with two x-points in the domain.

This is because INGRID identifies a configuration by decomposing subset of the domain contained within the limiter into three distinct regions: core, private-flux, and separatrix exterior.

We see this below with the core shaded in magenta, private-flux shaded in blue, and separatrix exterior with no filled shading.


Normalized Efit Data
1.5
1.0
0.5
Z
0.0
−0.5
−1.0
−1.5
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R
magx
xpt1
xpt2
limiter
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

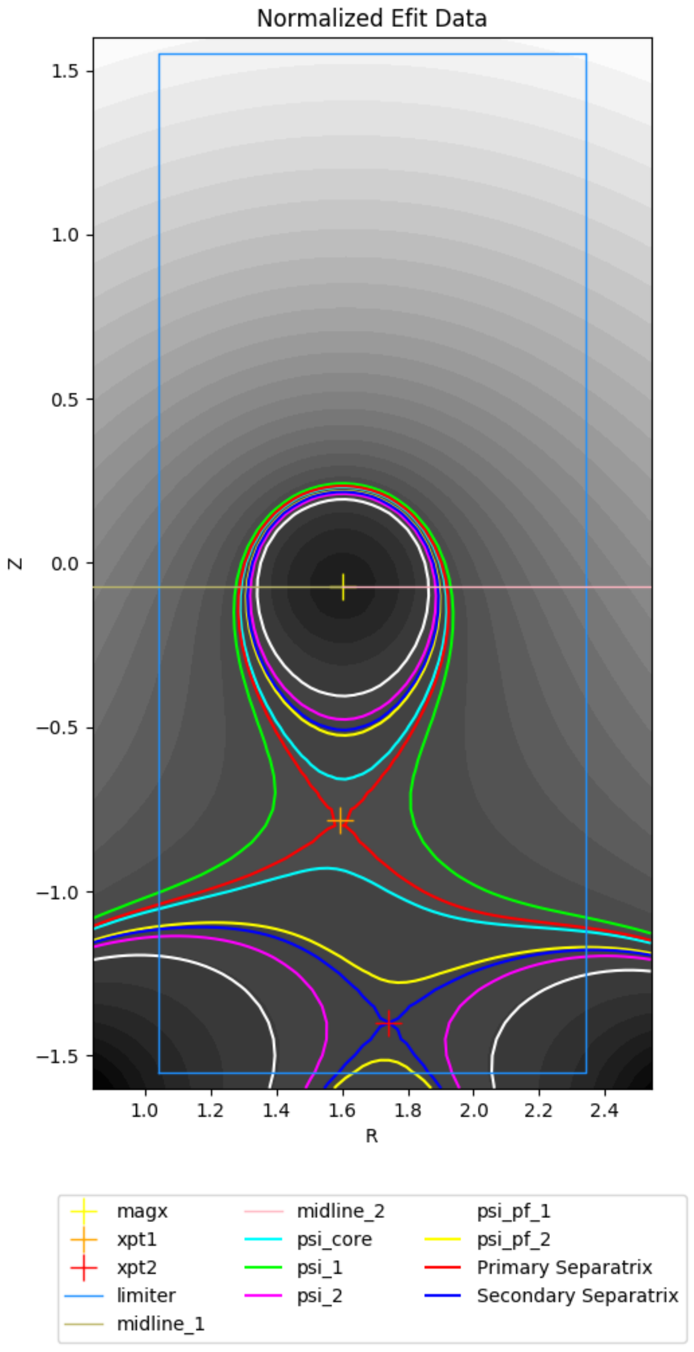
To emphasize the dependence on the limiter for decomposition, we also illustrate with setting both `efit_buffer_r` and `efit_buffer_z` back to their **default** values of `1.0e-2`.


Normalized Efit Data
1.5
1.0
0.5
Z 0.0
−0.5
−1.0
−1.5
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R
magx
xpt1
xpt2
limiter
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

It is up to the user to ensure the following is satisfied for successful identification of magnetic topology:

- The provided limiter forms a closed loop (note EFIT bounds do this by default and is therefore a useful tool for classification)
- The Magnetic-axis is contained within the closed limiter geometry
- The primary x-point is contained within the closed limiter geometry
- The primary separatrix “legs” intersect the limiter walls and form a closed region
- The secondary x-point is contained within the closed limiter geometry

We can indeed see that our parameter file has been preset to satisfy all of the above.



Normalized Efit Data
Z
R
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
magx
xpt1
xpt2
limiter
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

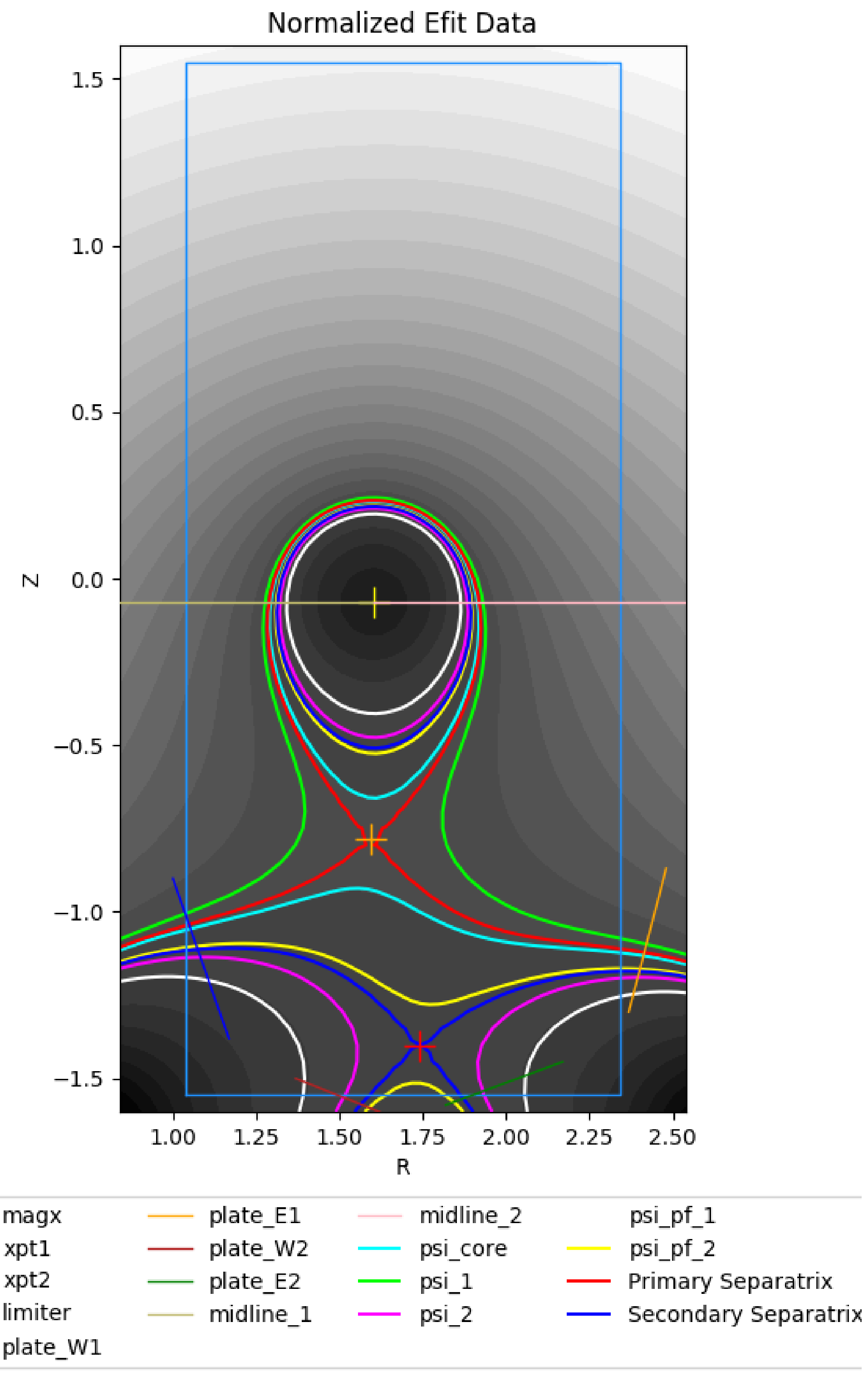
2.6.5 Using target-plates for generating a Patch map

The user can opt for using target-plates rather than a limiter for generating a Patch map for cases with two x-points in the domain. Setting the entry `strike_pt_loc` to `target_plates` tells INGRID to generate the Patch map using geometry provided in the parameter file block `target_plates`. We see this snippet below.

```
grid_settings:

  # ...
  # other settings
  # ...

  patch_generation:
    # ...
    # other settings
    # ...

    # strike_pt_loc takes the values of 'limiter' or 'target_plates'
    strike_pt_loc: target_plates  # generates a Patch map with a target_plates rather␣
↪than limiter block

# Specifications for using target plates
target_plates:

  # Target plate E of xpt1
  plate_E1:
    file: ../data/SF75/plate_E1.txt
    rshift: 1.0
    zshift: 0.28

  # Target plate E of xpt2
  plate_E2:
    file: ../data/SF75/plate_E2.txt
    rshift: 0.45
    zshift: 0.00

  # Target plate W of xpt1
  plate_W1:
    file: ../data/SF75/plate_W1.txt
    rshift: -0.2
    zshift: 0.2

  # Target plate W of xpt2
  plate_W2:
    file: ../data/SF75/plate_W2.txt
    zshift: 0.0
    rshift: 0.00
```

The user can refer to the diagrams earlier in the tutorial to see where the target plates above should reside in the domain.

By making the above edits, we then refresh our view of the data to obtain the following plot.


Normalized Efit Data
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Z
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R
magx
xpt1
xpt2
limiter
plate_W1
plate_E1
plate_W2
plate_E2
midline_1
midline_2
psi_core
psi_1
psi_2
psi_pf_1
psi_pf_2
Primary Separatrix
Secondary Separatrix

Indeed, we see there are now four target plates in the EFIT domain.

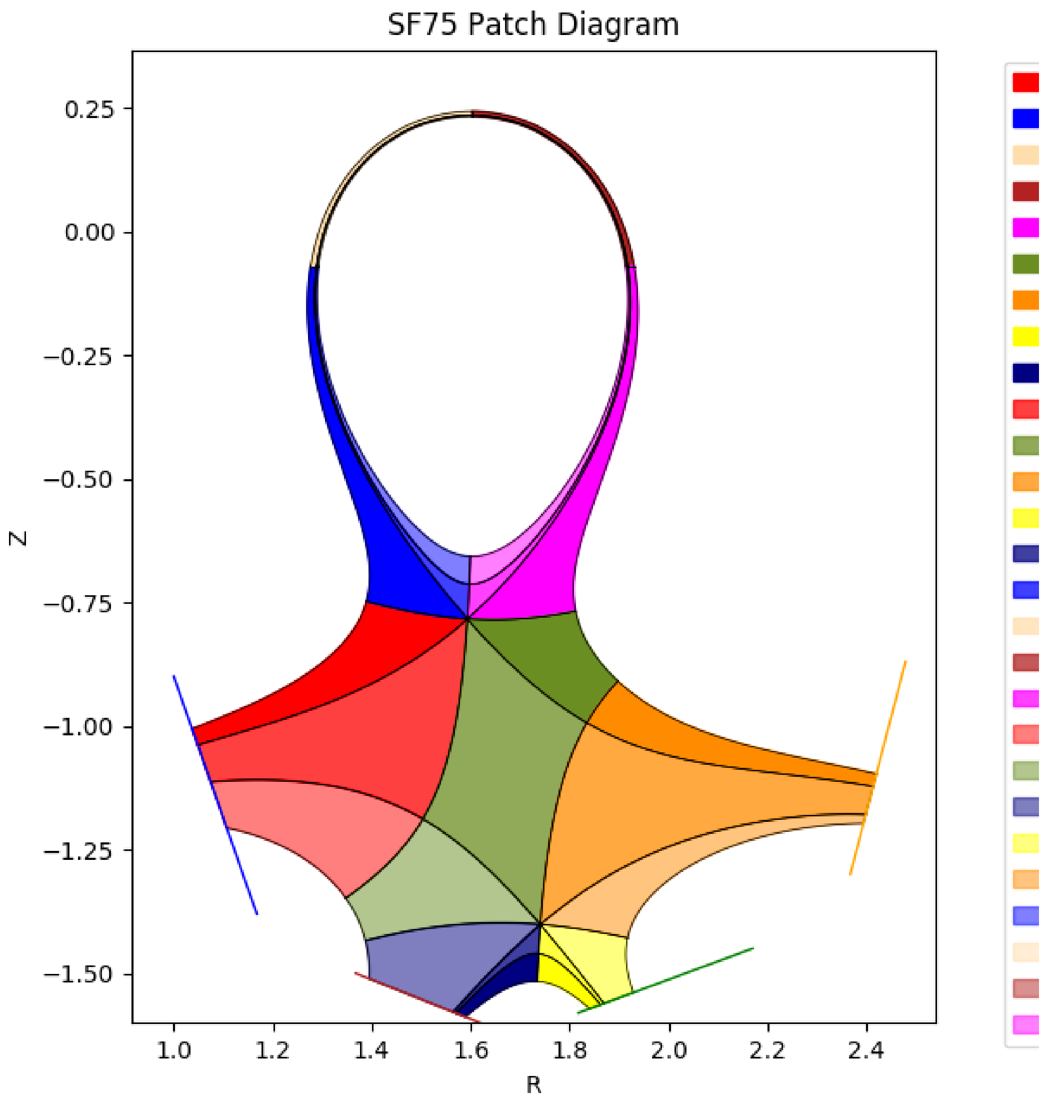
Note: **The presence of the limiter in the figure is not a bug.** Regardless of whether we decide to utilize the limiter or target plates for generating a Patch map, INGRID still relies on the limiter to classify the magnetic topology. This is why the user must be comfortable with limiter controls.

2.6.6 Creating the Patch map and grid

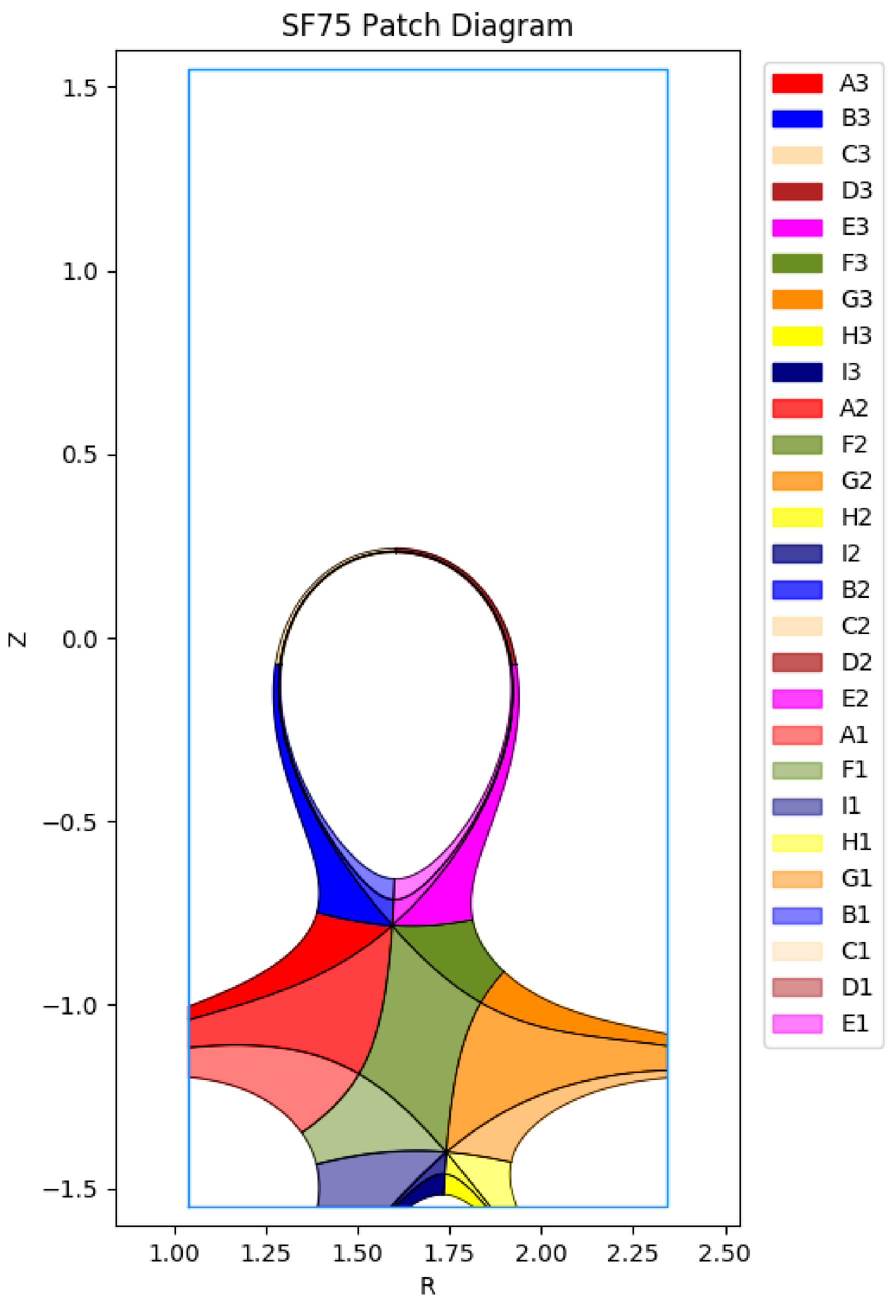
The majority of the effort for generating a Patch map goes into the preparation of the EFIT data. We indeed saw this in the previous sections. From here, we carry out the usual process of generating a Patch map that we saw in the SNL case.

All of the Patch map edit capabilities that we performed on the `SNL` cases are also available for all other cases with no further explanation.

We proceed with Patch map generation and obtain the following Patch map when keeping `strike_pt_loc` set with a value of `target_plates`.

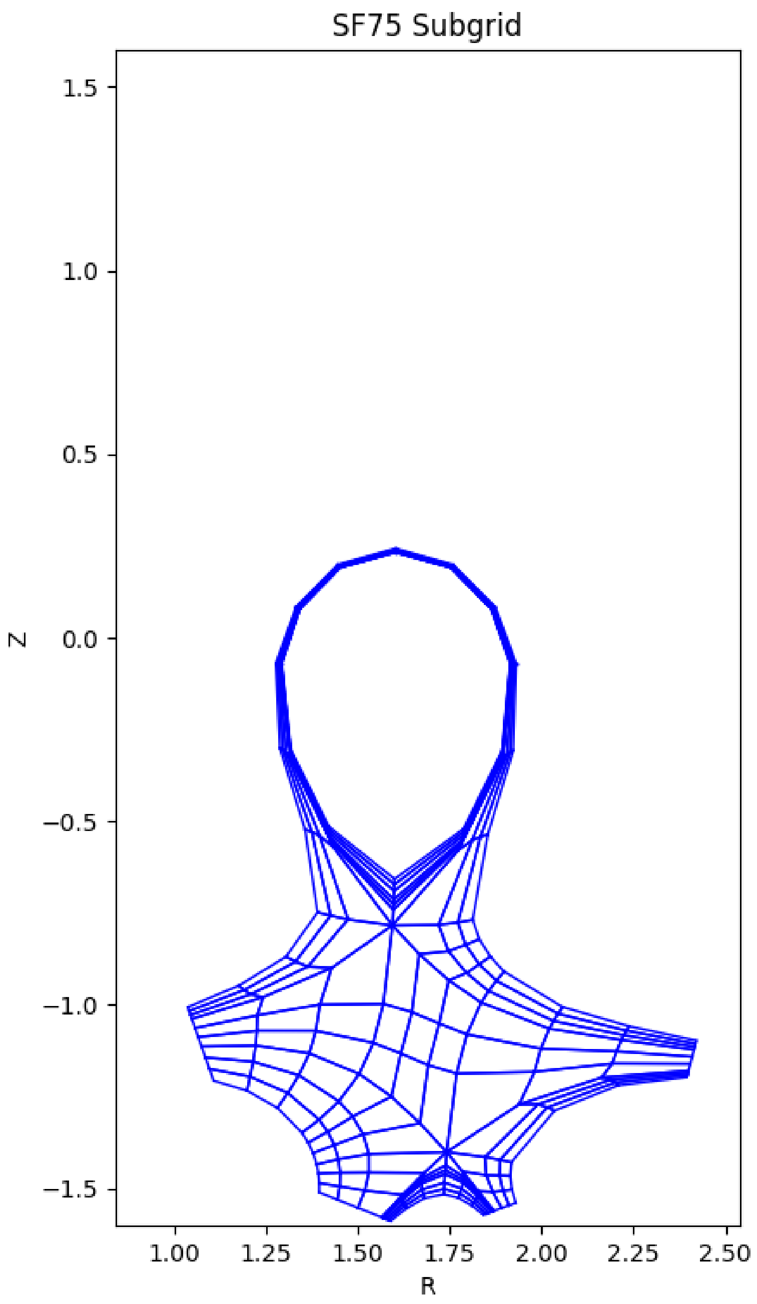

SF75 Patch Diagram
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
Z
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
R

If we had kept `strike_pt_loc` set to `limiter`, we would obtain the Patch map below.

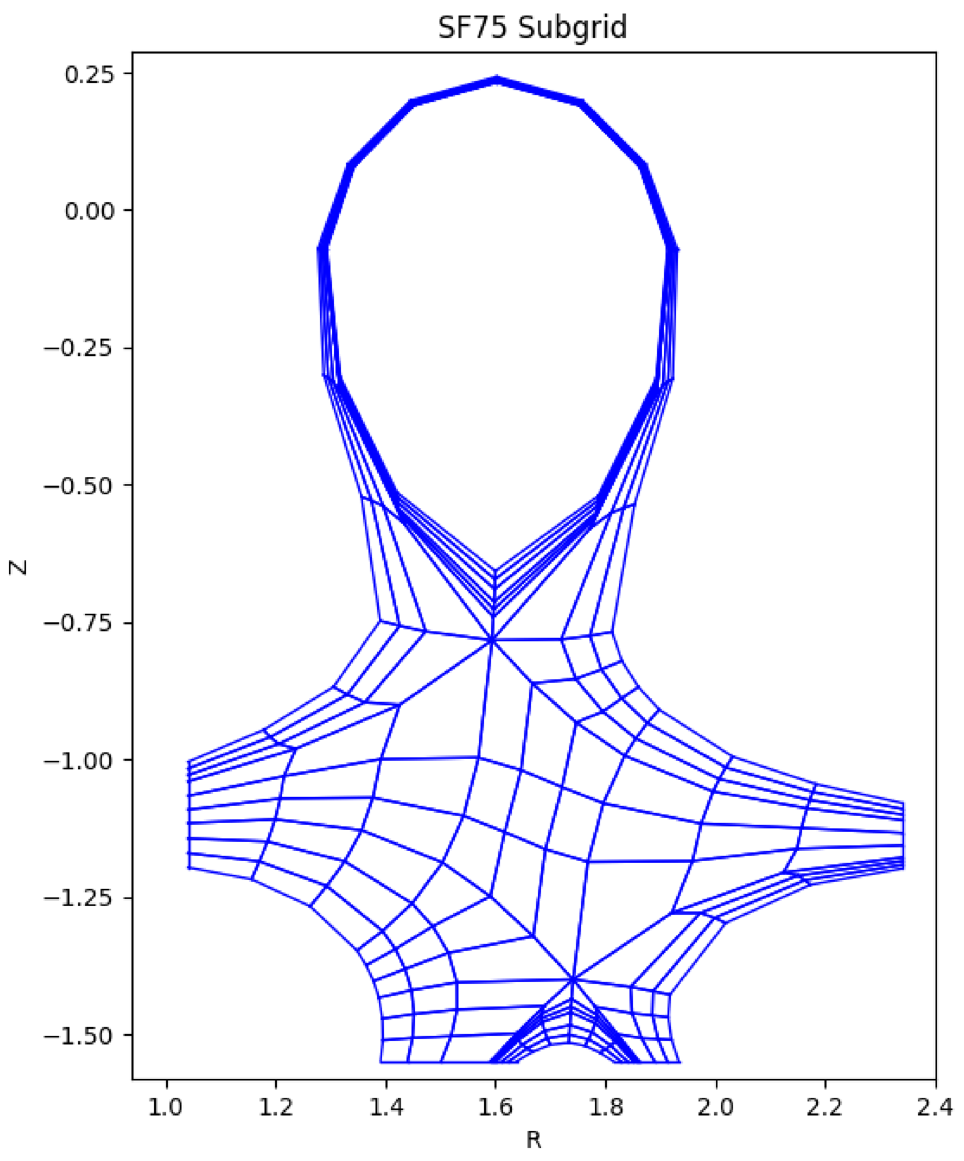

SF75 Patch Diagram
Z
R
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
1.00
1.25
1.50
1.75
2.00
2.25
2.50
A3
B3
C3
D3
E3
F3
G3
H3
I3
A2
F2
G2
H2
I2
B2
C2
D2
E2
A1
F1
I1
H1
G1
B1
C1
D1
E1

All grid generating instructions detailed in the SNL case also apply to the SF cases. The only difference is the addition of an additional control nr_3 that controls the Patch objects with name ending with 3.

Using np_default and nr_default with values of 3 produces the following grid when using target plates.


SF75 Subgrid
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
Z
1.00
1.25
1.50
1.75
2.00
2.25
2.50
R

Using `np_default` and `nr_default` with values of `3` produces the following grid when using the limiter.


SF75 Subgrid
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
Z
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
R

All other grid generation customization tools such as `distortion_correction`, `poloidal_f_A` - `poloidal_f_I`, and `radial_f_1` - `radial_f_3` are utilized in the same ways we saw in the earlier `SNL` cases.

CHAPTER

THREE

MODULE DOCUMENTATION

3.1 ingrid

Ingrid module for interfacing all grid generator capabilities.

This module contains the Ingrid class that drives all code functionality.

The Ingrid class is to encapsulate all functionality and allow the user to easily take advantage of advanced features of the code.

class `INGRID.ingrid.`**`Ingrid`**(*settings: dict = {}*, ***kwargs*)

Bases: *`INGRID.utils.IngridUtils`*

The driver class of the grid generator. A user will be able to load all experimental data, create patch maps, create grids, and export grids with the methods available. The GUI version of the Ingrid code is also accessed exclusively through this class.

Parameters

- **`settings`** (*`optional`*) – Dictionary representation of the settings file. The dictionary is obtained via a YAML dump while operating in gui mode. Providing no dictionary will populate the Ingrid object with a default value settings attribute. Any missing entries provided by a user will be populated with default values by Ingrid. (Refer to the Ingrid "Settings File Format" for a complete descri ption of the settings dictionary)
- **`**kwargs`** – Keyword arguments for processing of input data. (See method 'ProcessKeywords' in class 'IngridUtils' for a list of supported keywords)

Variables

- **`settings`** (*`dict`*) – Dictionary representation of the settings file.
- **`PlateData`** (*`dict`*) – Dictionary mapping target plates to their corresponding class 'Line' objects.
- **`LimiterData`** (`Line`) – Line class object representing tokamak limiter.
- **`magx`** (*`tuple`*) – (R, Z) coordinates of magnetic-axis (float entries).
- **`xpt1`** (*`tuple`*) – (R, Z) coordinates of primary x-point (float entries).
- **`xpt2`** (*`tuple`*) – (R, Z) coordinates of secondary x-point (float entries).
- **`PsiNorm`** (`EfitData`) – 'EfitData' class object containing normalized psi data calculated from class attribute 'PsiUNorm'.
- **`CurrentTopology`** (*`Topology`*) – Tokamak magnetic configuration type currently being operated on. Class 'Topology' acts as thebase class for all magnetic configurations.

`AnalyzeTopology()` → None
Perform analysis on normalized psi to determine magnetic topolgy.

`AutoRefineMagAxis()` → None
Refine magnetic axis RZ coordinates.

Will apply a root_finder method to 'rmagx' and 'zmagx' float values stored in attribute "settings['grid_settings']".

`AutoRefineXPoint()` → None
Refine primary x-point RZ coordinates.

Will apply a root_finder method to 'rxpt' and 'zxpt' float values stored in attribute "settings['grid_settings']".

`AutoRefineXPoint2()` → None
Refine secondary x-point RZ coordinates.

Will apply a root_finder method to 'rxpt2' and 'zxpt2' float values stored in attribute "settings['grid_settings']".

`CalcPsiNorm()` → None
Normalize psi data to a refined magnetic axis and primary x-point

`ClearLegend`(*ax*) → None
Safely remove the legend form the normalized psi data.

`ConstructGrid`(*NewFig: bool = True*, *ShowVertices: bool = False*) → None
Refine a generated patch map into a grid for exporting.

Parameters

- `NewFig` (*bool, optional*) – Plot the created grid on a new figure.
- `ShowVertices` (*bool, optional*) – Plot vertices in refined grid with bolded markers.
- `ListPatches` (*bool, optional*) – Generate a grid for a particular patch. Requires the correct patch name associated with the 'Patch' object.

  Warning: Grid generation is order dependent. Specifying a particular patch to generate a grid would only be done in rare cases and require the user to know dependencies for the particular patch.

`ConstructPatches()` → None
Create a patch map that can be refined into a grid.

This method assumes the user has either loaded patch data from a previous Ingrid session (see 'LoadPatches'), or that the user has already successfully called method 'AnalyzeTopology'.

Should the user want to automatically enable patch saving, the user should set the entry

'settings[''patch_data''][''preferences''][''new_file'']'

with a value of 'True'.

`CreatePatches()` → None
An alias for *ConstructPatches*. See *ConstructPatches* for more details.

`CreateSubgrid`(*NewFig: bool = True*, *ShowVertices: bool = False*) → None
Alias for *ConstructGrid*. See *ConstructGrid* for documentation.

`ExportGridue`(*fname: str = 'gridue'*, *guard_cell_eps=0.001*) → None
Export a gridue file for the created grid.

Parameters `fname` (*str, optional*) – Name of gridue file to save.

static `ImportGridue`(*fname: str = 'gridue'*) → dict

Import UEDGE grid file as dictionary.

Parameters **`fname`** (*`str, optional`*) – Path/file name to gridue formatted file.

Returns *A dict containing header and body information from the gridue file.*

`LoadEFIT`(*fpath: str*) → None

`LoadGeometryData`(*geo_items: dict*) → None

Load strike geometry RZ coordinates from external file.

Said external file must be of type '.txt' or a properly formatted '.npy' file.

Parameters **`geo_items`** (*`dict`*) – The argument dict specifying which strike geometry to load data into.

Notes

`geo_items` is of the form: `{geo_name:  data_fname, ... }`

where both `geo_name` and `data_fname` are of type str.

Multiple `data_fname` files can be loaded at once in a manner similar to method *`SetGeometry()`*.

Raises

- **`ValueError`** – If 'geo_items' is not type 'dict'.
- **`ValueError`** – If invalid 'geo_items' key provided.
- **`ValueError`** – If data_fname entry is not of type 'str'.
- **`ValueError`** – If data_fname is not a file.
- **`ValueError`** – If data_fname file is not of format '.txt' or '.npy'

`LoadPatches`(*fname: str = ''*) → None

Load patches stored in an Ingrid generated '.npy' file.

Parameters **`fname`** (*`str, optional`*) –

Path to patch data. If no fname is provided to method 'LoadPatches', Ingrid code will check the settings 'dict' for a file under entry `settings['patch_data']['file']`

`PlotEastWestXpt1Ref`(*ax: Optional[object] = None*) → None

Plot midplane line through magnetic axis with any applied transformations specified in settings.

This method can be used to inspect the effects of 'magx_tilt_1', 'magx_tilt_2', 'rmagx_shift', and 'zmagx_shift'.

`PlotEastWestXpt2Ref`(*ax: Optional[object] = None*) → None

Plot midplane line through magnetic axis with any applied transformations specified in settings.

This method can be used to inspect the effects of 'magx_tilt_1', 'magx_tilt_2', 'rmagx_shift', and 'zmagx_shift'.

`PlotGrid`() → None

Plot the grid that was generated with method 'ConstructGrid'.

static `PlotGridue`(*GridueParams: dict*, *edgecolor='black'*, *ax: Optional[object] = None*)

Plot UEDGE grid from 'dict' obtained from method 'ImportGridue'

Parameters

- **GridueParams** (*dict*) – Gridue header and body information as a dictionary. (See method ImportGridue)
- **edgecolor** (*str, optional*) – Color of grid.
- **ax** (*object, optional*) – Matplotlib axes to plot on.

PlotLimiter(*ax: Optional[object] = None*) → None
Plot limiter geometry.

PlotMidplane(*ax: Optional[object] = None*) → None
Plot midplane line through magnetic axis with any applied transformations specified in settings.

This method can be used to inspect the effects of 'magx_tilt_1', 'magx_tilt_2', 'rmagx_shift', and 'zmagx_shift'.

PlotPatches() → None
Plot the patch map that was generated with method 'CreatePatches'

PlotPsiLevel(*efit_psi: object*, *level: float*, *Label: str = ''*) → None
Plot a contour corresponding to a psi level.

Parameters

- **efit_psi** (EfitData) – The 'EfitData' object to get the psi data from.
- **level** (*float*) – The psi value to plot.
- **Label** (*str, optional*) – Label to provide to matplotlib.pyplot.contour.

PlotPsiNorm(*view_mode: str = 'filled'*) → None
Plot normalized psi data.

PlotPsiNormBounds() → None
Plot contour lines associated with psi boundary values provided.

This method extracts psi values from the 'settings' dict and plots the psi level. In addition to the psi values in 'settings', the primary and, if applicable, secondary separatrix are plotted as well.

If the user is operating on a single null configuration, the psi values plotted are 'psi_1', 'psi_core', 'psi_pf_1'.

If the user is operating on a case with two x-points, the psi values are the same as above but with 'psi_1', 'psi_2', and 'psi_pf_2' also included in the plot.

PlotPsiNormMagReference(*ax: Optional[object] = None*) → None
Plot a marker on the magnetic axis and all x-points of interest.

PlotPsiUNorm() → None
Plot unnormalized psi data.

PlotStrikeGeometry(*ax: Optional[object] = None*) → None
Plot all strike geometry to be used for drawing the Patch Map.

Checks the central INGRID `settings` attribute for whether `settings['patch_generation']['strike_pt_loc']` is `True` or if `settings['grid_settings']['num_xpt']` is equal to 2 in order to determine whether or not to plot the limiter geometry.

Otherwise only any loaded target plates will be plotted.

PlotSubgrid() → None
Alias for method *PlotGrid*. See *PlotGrid* for documentation.

PlotTargetPlate(*plate_key: str*, *color: str = 'red'*, *ax: Optional[object] = None*) → None
Plot a target plate corresponding to a plate key.

Parameters

- **plate_key** (*str*) – An Ingrid supported target plate key (see method SetGeometry for supported plate keys)
- **color** (*str, optional*) – Color to provide to matplotlib

PlotTargetPlates(*ax: Optional[object] = None*) → None
Plot all PlateData and remove outdated plate line artists.

PlotTopologyAnalysis() → None
Shade the private flux, core, and show where the secondary x-point travels.

This method can be used to interpret which type of configuration the user is handling.

PrintSummaryInput() → None
Print a summary of the currently loaded data files

Will print relevant EQDSK, patch data files, target plate files, and limiter files.

PrintSummaryParams() → None
Print a summary of key settings values.

static **ReadYamlFile**(*FileName: str*) → dict
Read a yaml file and return a dictionary

Parameters **FileName** (*str*) – Path/file name of '.yml' parameter file represented as dictionary.

Returns *Settings file represented as a dictionary*

Raises **IOError** – If error occurs while loading yml file.:

RefreshSettings()

RemovePlotLine(*label: str*, *ax: Optional[object] = None*) → None

RemovePlotPatch(*label: str*, *ax: Optional[object] = None*) → None

RemovePlotPoint(*label: str*, *ax: Optional[object] = None*) → None

SaveGeometryData(*geo_items: dict*, *timestamp: bool = False*) → None
Save strike geometry Line object RZ coordinates as '.npy' file.

Geometry data files created with this method can be used in the INGRID parameter file.

Parameters

- **geo_items** (*dict*) – A dictionary specifying which target plate to save and the file name/path. Said dictionary takes the following form:

  ```
  {geo_name:  data_fname, ... }
  ```

  Where both geo_name and data_fname are of type str. Multiple data_fname files can be saved at once in a manner similar to method 'SetGeometry' (see documentation for 'SetGeometry').
- **timestamp** (*bool, optional*) – Append a time stamp to the end of the files.

Raises

- **ValueError** – If 'geo_items' is not type 'dict'.
- **ValueError** – If invalid 'geo_items' key provided.
- **ValueError** – If data_fname entry is not of type 'str'.
- **ValueError** – If data_fname entry is an empty string.

- `ValueError` – If requested strike geometry Line to save has no data.

`SavePatches`(*fname: str = ''*) → None

Save patches as '.npy' file for later reconstruction in Ingrid.

This file contains the information required to reconstruct patches at a later time and bypass the line_tracing.

Parameters `fname` (*str, optional*) – Name of file/location for patch data.

`SaveSettingsFile`(*fname: str = ''*, *settings: dict = {}*) → pathlib.Path

Save a new settings .yml file.

Parameters

- `fname` (*optional*) – Name of new settings '.yml' file. If default value of '', then Ingrid will generate a '.yml' file named 'INGRID_Session' appended with a timestamp.
- `settings` (*optional*) – Ingrid settings dictionary to dump into the '.yml' file. Defaults to empty dict which produces a template settings file.

Returns *A Path instance representing the saved YAML file.*

`SetGeometry`(*geo_items: dict*, *rshift: float = 0.0*, *zshift: float = 0.0*) → None

Define and load the tokamak strike geometry into the Ingrid object. Allows the user to set target plate data and/or limiter data that will be used to generate a patch map.

Parameters

- `geo_items` (*dict*) – Argument dict specifying which item(s) to set and by what way.
- `rshift` (*float, optional*) – Translate 'geo_items' to coordinate R', with R' = R + rshift. Will override all 'rshift' provided entries in 'geo_items'.
- `zshift` (*float, optional*) – Translate 'geo_items' to coordinate Z', with Z' = Z + zshift. Will override all 'zshift' provided entries in 'geo_items'.

Notes

Multiple geometry items can be set at once, but the following key-value format must be obeyed for any number of entries:

`{geo_key: geo_item}`

All keys for 'geo_items' are type 'str'. Accepted key values are as follows:

Geometry	Accepted Keys (str)
Plate W1	`plate_W1`, `W1`
Plate E1	`plate_E1`, `W1`
Plate W2	`plate_W2`, `W1`
Plate E2	`plate_E2`, `W1`
Limiter	`limiter`, `wall`

The above keys are NOT case sensitive.

Corresponding key values can vary in data type depending on means of setting geometry. The types and their usage are as follows:

'str': Path to '.txt' file containing coordinate data or to Ingrid generated '.npy' file (obtained via methods

'SaveLimiterData' and 'SaveTargetPlateData'). Note: When setting Limiter geometry, the user can provide the value 'default' to set limiter data to that which is contained in the loaded neqdsk file.

'list', 'tuple': Iterables provided as values must be of length == 2. The first entry corresponds to R coordinate information, and the second entry corresponds to Z coordinate information. This information can be in the form of a list or NumPy array with shape == (N,).

'dict': One can map to a dictionary taking on a variety of formats depending on need.

Setting geometry with explicit RZ coordinates: {'R': R_data, 'Z': z_data} Where R_data and Z_data are in the form of a list or NumPy array with shape == (N,) as above.

Setting geometry with data from external file: {'file': str}

Notes: The above dict option support keys 'rshift' and 'zshift' for the user to provide transformations to individual geometry items (see examples).

Because the core 'settings' attribute contained by the Ingrid class contains dict structures itself, the user can also provide settings['limiter'] and settings['target_plates'][k] to method 'SetGeometry' (where k corresponds to a plate key).

Examples

Setting default limiter data contained in loaded neqdsk file:

```
>>> MyIG = Ingrid()
>>> MyIG.SetGeometry({'limiter': 'default'})
```

Setting target plate 'E1' with numpy array:

```
>>> MyIG = Ingrid()
>>> MyIG.SetGeometry({'E1': 'E1_data.npy'})
```

Setting limiter data with Ingrid '.npy' file:

```
>>> MyIG = Ingrid()
>>> MyIG.SetGeometry({'limiter': 'LimiterData.npy'})
```

Setting both target plates 'W1' and 'E1' with '.txt' and '.npy' files while only applying rshift and zshift to target plate 'E1' (Note the dict structure used for specifying 'E1'):

```
>>> MyIG = Ingrid()
>>> geometry_dict = {
... 'W1': 'W1_data.txt',
... 'E1': {
... 'file': 'E1_data.npy',
... 'rshift': 1.23, 'zshift': 3.14
... }
... }
>>> MyIG.SetGeometry(geometry_dict)
```

Setting plate 'W2' with user provided NumPy array for RZ coordinates:

```
>>> MyIG = Ingrid()
>>> R_data = my_numpy_array_1
>>> Z_data = my_numpy_array_2
>>> RZ_dict = {'R': my_numpy_array_1, 'Z': my_numpy_array_2}
>>> MyIG.SetGeometry({'plate_W1': RZ_dict})
```

Raises

- `ValueError` – If file path provided does not lead to actual file.
- `ValueError` – If file provided is not of suffix '.txt' or '.npy'.
- `ValueError` – If invalid 'geo_items' key provided.
- `ValueError` – If 'geo_items' dict value contains invalid key.
- `ValueError` – If value associated with 'geo_items' key is not of supported data type.

`SetMagReference()` → None
Set the appropriate reference points in the domain. Namely the magnetic-axis, primary x-point, and (if applicable), secondary x-point.

`SetTargetPlates()` → None
Define target plate geometries based off of `settings` dict specifications.

This is a convenience method that calls SetGeometry.

`SetTopology(`*topology: str*`)` → None
Initialize the current topology to a particular magnetic topology.

Parameters `topology` (*str*) – String literal corresponding to which magnetic topology to initialize.

Values can be: 'LSN': Lower Single Null 'USN': Upper Single Null 'UDN': Unbalanced Double Null 'SF15': Snowflake-15 'SF45': Snowflake-45 'SF75': Snowflake-75 'SF105': Snowflake-105 'SF135': Snowflake-135 'SF165': Snowflake-165

Raises `ValueError` – If user provided unrecognized string entry.

`ShowSetup(`*view_mode: str = 'filled'*`)` → None
Show Ingrid setup that a patch map will be generated from.

This method plots normalized psi data, psi boundaries, strike geometry, and midplane lines through the magnetic axis.

`StartGUI()` → None
Start GUI for Ingrid.

Will assume usage on a machine with tk GUI capabilities. No prior settings file is required as the user will be prompted with an option to generate a new file.

`StartSetup(`***kwargs*`)` → None
A collection of essential tasks before generating a patch map from scratch.

The user should ensure that the 'settings' dict is populated with the correct paths to relevant neqdsk data and geometry files.

The user should ensure 'settings[''grid_settings''][''num_xpt'] is set with the correct integer value.

`INGRID.ingrid.QuickStart()` → None
Start Ingrid in gui mode with default settings.

Rather than providing data to the class constructor, a user can opt to start Ingrid immediately in it's gui form. This is useful for users who are not familiar with the Ingrid class and it's capabiliites. Advanced users may still find QuickStart useful, but would also use the code in self-authored scripts.

3.2 topologies

The `topologies` subpackage contains modules for INGRID supported grid topologies.

3.2.1 `topologies.sf105`

The `sf105` module contains *SF105* for representing a Snowflake-105 topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.sf105.`**`SF105`**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF105'*)

Bases: *`INGRID.utils.TopologyUtils`*

The *SF105* class for handling *Snowflake-105* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the SF105 object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** (*`dict`*) – The collection of Patch objects representing the topology.

`AdjustGrid`() → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **`patch`** (`Patch`) – The patch to tidy up (will only adjust if next to x-point).

`AdjustPatch`(*patch*)

`GroupPatches`()

`OrderPatches`()

`construct_patches`()

`set_gridue`()

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.2 `topologies.sf135`

The `sf135` module contains *SF135* for representing a Snowflake-135 topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.sf135.`**`SF135`**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF135'*)

Bases: *`INGRID.utils.TopologyUtils`*

The *SF135* class for handling *Snowflake-135* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the SF135 object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** (*`dict`*) – The collection of Patch objects representing the topology.

`AdjustGrid``()` → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **`patch`** (`Patch`) – The patch to tidy up (will only adjust if next to x-point).

`AdjustPatch`(*patch*)

`GroupPatches``()`

`OrderPatches``()`

`construct_patches``()`

Draws lines and creates patches for both USN and LSN configurations.

Patch Labeling Key: I: Inner, O: Outer, DL: Divertor Leg, PF: Private Flux, T: Top, B: Bottom, S: Scrape Off Layer, C: Core.

`set_gridue``()`

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.3 `topologies.sf15`

The `sf15` module contains *`SF15`* for representing a Snowflake-15 topology/configuration.

Child of base `utils.TopologyUtils`.

`class INGRID.topologies.sf15.`**`SF15`**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF15'*)

Bases: *`INGRID.utils.TopologyUtils`*

The *SF15* class for handling *Snowflake-15* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the SF15 object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** (*`dict`*) – The collection of Patch objects representing the topology.
- **`ConnexionMap`** – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** – The collection of Patch objects representing the topology.

AdjustGrid() → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **patch** (Patch) – The patch to tidy up (will only adjust if next to x-point).

AdjustPatch(*patch*)

GroupPatches()

OrderPatches()

construct_patches()

set_gridue()

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.4 topologies.sf165

The `sf165` module contains *SF165* for representing a Snowflake-165 topology/configuration.

Child of base `utils.TopologyUtils`.

class INGRID.topologies.sf165.**SF165**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF165'*)

Bases: *INGRID.utils.TopologyUtils*

The *SF165* class for handling *Snowflake-165* configurations within a tokamak.

Parameters

- **Ingrid_obj** (Ingrid) – Ingrid object the SF165 object is being managed by.
- **config** (*str, optional*) – String code representing the configuration.

Variables

- **ConnexionMap** (*dict*) – A mapping defining dependencies between Patch objects for grid generation.
- **patches** (*dict*) – The collection of Patch objects representing the topology.

AdjustGrid() → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **patch** (Patch) – The patch to tidy up (will only adjust if next to x-point).

AdjustPatch(*patch*)

GroupPatches()

OrderPatches()

construct_patches()

Draws lines and creates patches for both USN and LSN configurations.

Patch Labeling Key: I: Inner, O: Outer, DL: Divertor Leg, PF: Private Flux, T: Top, B: Bottom, S: Scrape Off Layer, C: Core.

set_gridue()

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.5 `topologies.sf45`

The `sf45` module contains *`SF45`* for representing a Snowflake-45 topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.sf45.`**`SF45`**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF45'*)

Bases: *`INGRID.utils.TopologyUtils`*

The *SF45* class for handling *Snowflake-45* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the SF45 object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** (*`dict`*) – The collection of Patch objects representing the topology.

`AdjustGrid``()` → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **`patch`** (`Patch`) – The patch to tidy up (will only adjust if next to x-point).

`AdjustPatch`(*patch*)

`GroupPatches``()`

`OrderPatches``()`

`construct_patches``()`

`set_gridue``()`

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.6 `topologies.sf75`

The `sf75` module contains *`SF75`* for representing a Snowflake-75 topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.sf75.`**`SF75`**(*Ingrid_obj: ingrid.Ingrid*, *config: str = 'SF75'*)

Bases: *`INGRID.utils.TopologyUtils`*

The *SF75* class for handling *Snowflake-75* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the SF75 object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.

- **patches** (*dict*) – The collection of Patch objects representing the topology.

AdjustGrid() → None
Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **patch** (Patch) – The patch to tidy up (will only adjust if next to x-point).

AdjustPatch(*patch*)

GroupPatches()

OrderPatches()

construct_patches()

set_gridue()
Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.7 topologies.snl

The `snl` module contains *SNL* for representing a single-null topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.snl.`**SNL**(*Ingrid_obj: ingrid.Ingrid*, *config: str*)
Bases: *INGRID.utils.TopologyUtils*

The SNL class for handling *Lower Single Null* (LSN) and *Upper Single Null* (USN) configurations within a tokamak.

Parameters

- **Ingrid_obj** (Ingrid) – Ingrid object the SNL object is being managed by.
- **config** (*str*) – String code representing the configuration (for SNL it can be 'LSN' or 'USN').

Variables

- **ConnexionMap** (*dict*) – A mapping defining dependencies between Patch objects for grid generation.
- **patches** (*dict*) – The collection of Patch objects representing the topology.

AdjustGrid() → None
Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **patch** (Patch) – The patch to tidy up (will only adjust if next to x-point).

AdjustPatch(*patch*)

GroupPatches()

OrderPatches()

construct_patches()
Create the Patch map with `LineTracing`.

set_gridue()
Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.2.8 `topologies.udn`

The `udn` module contains *UDN* for representing an unbalanced double-null topology/configuration.

Child of base `utils.TopologyUtils`.

class `INGRID.topologies.udn.`**`UDN`**(*Ingrid_obj: ingrid.Ingrid*, *config:* UDN)

Bases: *`INGRID.utils.TopologyUtils`*

The *UDN* class for handling *Unbalanced Double Null* configurations within a tokamak.

Parameters

- **`Ingrid_obj`** (`Ingrid`) – Ingrid object the UDN object is being managed by.
- **`config`** (*`str, optional`*) – String code representing the configuration.

Variables

- **`ConnexionMap`** (*`dict`*) – A mapping defining dependencies between Patch objects for grid generation.
- **`patches`** (*`dict`*) – The collection of Patch objects representing the topology.

`AdjustGrid`() → None

Adjust the grid so that no holes occur at x-points, and cell grid faces are alligned

A small epsilon radius is swept out around x-points during Patch line tracing. This simple tidies up a grid.

Parameters **`patch`** (`Patch`) – The patch to tidy up (will only adjust if next to x-point).

`AdjustPatch`(*patch*)

`GroupPatches`()

`OrderPatches`()

`construct_patches`()

`set_gridue`() → dict

Prepares a `gridue_settings` dictionary with required data for writing a gridue file.

3.3 utils

Helper classes for the ingrid module and topologies package.

This module contains classes *IngridUtils* and *TopologyUtils*. These classes encapsulate much of the critical methods for handling file I/O, topology analysis, generating patch maps, and generating grids.

class `INGRID.utils.`**`IngridUtils`**(*settings: dict = {}*, ***kwargs*)

Bases: `object`

The base class for `ingrid.Ingrid` that handles backend management of key Ingrid capabilities. This class can be directly utilized by advanced users and developers of the Ingrid code.

Class *IngridUtils* encapsulates implementation details of file I/O, keyword parsing, sorting of geometry, and managing of other helper classes such as `line_tracing.LineTracing` and `interpol.EfitData`.

Parameters

- **settings** (*optional*) – Dictionary representation of the settings file. The dictionary is obtained via a YAML dump while operating in gui mode. Providing no dictionary will populate the Ingrid object with a default value settings attribute. Any missing entries provided by a user will be populated with default values by Ingrid. (Refer to the Ingrid "Settings File Format" for a complete descri ption of the settings dictionary)
- ****kwargs** – Keyword arguments for processing of input data. (See method 'ProcessKeywords' in class 'IngridUtils' for a list of supported keywords)

Variables

- **InputFile** (*str*) – Path to the parameter file.
- **config** (*str*) – The configuration of the topology.
- **settings** (*dict*) – Core settings dictionary containing all data used for generating patches and grids.
- **settings_lookup** (*dict*) – Top level entries of the parameter file and convenience attribute for later accessing of entries.
- **default_values_lookup** (*dict*) – General structure of YAML settings file in dictionary form.
- **default_grid_settings** (*dict*) – Default entries for key *grid_settings* in the YAML settings file.
- **default_integrator_settings** (*dict*) – Default entries for key *integrator_settings* in the YAML settings file.
- **default_target_plate_settings** (*dict*) – Default entries for key *target_plates* in the YAML settings file.
- **default_limiter_settings** (*dict*) – Default entries for key *limiter* in the YAML settings file.
- **default_patch_data_settings** (*dict*) – Default entries for key *patch_data* in the YAML settings file.
- **default_DEBUG_settings** (*dict*) – Default entries for key *DEBUG* in the YAML settings file.
- **PlateData** (*dict*) – Dictionary containing Line objects that correspond to target plates loaded by the user.
- **OMFIT_psi** (*OMFITgeqdsk*) – Instance of class used to interface G files generated by EFIT.
- **PsiUNorm** (EfitData) – *EfitData* class object containing unormalized psi data from provided neqdsk file in settings.
- **LimiterData** (Line) – Line class object representing tokamak limiter.
- **magx** (*tuple*) – (R, Z) coordinates of magnetic-axis (float entries).
- **xpt1** (*tuple*) – (R, Z) coordinates of primary x-point (float entries).
- **xpt2** (*tuple*) – (R, Z) coordinates of secondary x-point (float entries).
- **LineTracer** (LineTracing) – *LineTracing* instance that for topology analysis and poloidal/radial tracing.

CheckPatches(*verbose: bool = False*) → None

Check that Patch objects adjacent to target plates are monotonic in psi.

This method is a wrapper for the CheckPatches method specific to the current topology being operated on.

Parameters `verbose` (*bool, optional*) – Flag for printing full output to terminal. Defaults to False.

`ClassifyTopology`(*visual=False*) → None

Determine the topology that is being operated on.

Inspects the `settings['grid_settings']['num_xpt']` entry to determine which classification scheme to employ.

Parameters `visual` (*bool, optional*) – Flag for activating the analysis of x-points in a visual DEBUG mode. Default is False.

Raises `ValueError` – If user specifies `settings['grid_settings']['num_xpt']` with value other than 1 (int) or 2 (int).

`FindMagAxis`(*r: float*, *z: float*) → None

Refine the entries and assign to the magnetic-axis in settings.

Parameters

- `r` (*float*) – R coordinate of magnetic-axis guess.
- `z` (*float*) – Z coordinate of magnetic-axis guess.

`FindXPoint`(*r: float*, *z: float*) → None

Refine the entries and assign to the primary x-point in settings.

Parameters

- `r` (*float*) – R coordinate of primary x-point guess.
- `z` (*float*) – Z coordinate of primary x-point guess.

`FindXPoint2`(*r: float*, *z: float*) → None

Refine the entries and assign to the secondary x-point in settings.

Parameters

- `r` (*float*) – R coordinate of secondary x-point guess.
- `z` (*float*) – Z coordinate of secondary x-point guess.

`GetMagxData`() → tuple

Return the magnetic-axis (r,z) coordinates and associated un-normalized psi value.

Returns *A 3-tuple of r-z coordinates and scalar psi value*

`GetPatchTagMap`(*config: str*) → dict

Get the Patch-Tag mapping for a particular configuration.

This mapping is used to identify patch names with a particular patch tag code.

Parameters `config` (*str*) – The configuration to get the patch tag map for.

Returns *A dictionary containing the tag to patch name mappings.*

`GetXptData`() → dict

Return all x-point (r,z) coordinates and associated un-normalized psi values.

Returns *A dict containing an (r, z, psi) entry for each x-point*

`OMFIT_read_psi`() → None

Python class to read the psi data in from an ascii file.

Saves the boundary information and generates an EfitData instance.

OrderLimiter() → None

Ensures the limiter points are oriented **clockwise** around the magnetic axis (per INGRID convention).

This method requires the limiter geometry to have been defined as well as the magnetic axis to be refined.

Notes

Ordering of limiter is **crucial** when using limiter for creating a patch map. This occurs for all cases with two x-points.

OrderTargetPlate(*plate_key: str*) → None

Ensures the target plate points are oriented **clockwise** around the magnetic axis (per INGRID convention).

Parameters **plate_key** (*str*) – The key corresponding to target plate Line object to sort.

Notes

Ordering of target plates is **crucial** when using target plates for creating a patch map.

Valid plate keys are as follows:

Plate	Accepted Keys (str)
Plate W1	`plate_W1`, `W1`
Plate E1	`plate_E1`, `E1`
Plate W2	`plate_W2`, `W2`
Plate E2	`plate_E2`, `E2`

OrderTargetPlates() → None

Convenience method for ordering target plate Point objects.

ParseTxtCoordinates(*fpath: str*, *rshift: float = 0.0*, *zshift: float = 0.0*) → tuple

Extract the (R,Z) coordinates from a .txt file.

Files of types .txt must conform to the following format:

$$r_0, z_0$$
$$r_1, z_1$$
$$r_2, z_2$$
$$.....$$
$$r_n, z_n$$

Put otherwise, r and z values are differentiated by a comma and each coordinate must appear on a new line withing the file.

If a line starts with the character '#', it will be skipped.

Parameters

- **fpath** (*str*) – The path to the text file containing (R, Z) coordinate entries.
- **rshift** (*float, optional*) – Applies a translation to the R coordinate of the text file entries.
- **zshift** (*float, optional*) – Applies a translation to the Z coordinate of the text file entries.

Returns *A 2-tuple (R, Z) with list entries containing R and Z data respectively.*

Raises

- `IOError` – If error occurs while reading text file.
- `ValueError` – If fpath string provided leads to an invalid file.

`PopulateSettings`(*settings: dict*, *verbose: bool = True*) → None
: Populate a settings dict with any missing entries that INGRID may need.

This should be used to screen for any illegal parameter file entries and to ensure clean data entry.

Parameters

- **settings** (*dict*) – Dictionary object conforming to structure of `settings` dictionary
- **verbose** (*bool, optional*) – Print full output to terminal. Defaults to False.

`PrepGridue`(*guard_cell_eps=0.001*) → None
: Prepare the gridue for writing.

This method calls topology specific implementations of methods that concatenate the Patch object subgrids into a global grid.

`PrepLineTracing()`
: Initializes the line tracing class for the construction of the grid.

`ProcessKeywords`(***kwargs*) → None
: Process kwargs and set all file paths accordingly.

`ProcessPaths()` → None
: Update settings by pre-pending path entries to all file entries.

`ReconstructPatches`(*raw_patch_list: list*) → dict
: Reconstruct a Patch objects from a saved file.

This method takes in an Ingrid formatted .npy file that contains the information needed to reconstruct a patch map from a past INGRID session.

Parameters **fname** (*str*) – The file path to the patch data file obtained after a call to Ingrid class method SavePatches

Returns *A dict of reconstructed Patch objects.*

`SetDefaultSettings()` → None
: Set all default values that will populate the `settings` dict.

This instantiates the following entries within the settings file:

- ‘grid_settings’
- ‘integrator_settings’
- ‘target_plates’
- ‘limiter’
- ‘patch_data’
- ‘DEBUG’

Additional entries may be added here as development continues.

`SetLimiter`(*fpath: str = ''*, *coordinates: list = []*, *rshift: float = 0.0*, *zshift: float = 0.0*) → None
: Instantiate the class Line object that represents the tokamak limiter.

This method accepts either coordinates or a valid file path to coordinate data.

If *fpath* and *coordinates* are at their default values, then the EFIT data will be searched for it's default limiter values.

Parameters

- **fpath** (*str, optional*) – A file path to a '.txt' or '.npy' file containing (R, Z) data.
- **coordinates** (*list, optional*) – A list with two entries containing R and Z entries respectively.
- **rshift** (*float, optional*) – Apply a translation to the R coordinate of the limiter entries.
- **zshift** (*float, optional*) – Apply a translation to the Z coordinate of the limiter entries.

`SetTargetPlate`(*settings: dict*, *rshift: float = 0.0*, *zshift: float = 0.0*) → None

Initialize a target plate Line object.

This method can initialize target plates:

- *plate_W1*
- *plate_E1*
- *plate_W2*
- *plate_E2*

with explicit (R, Z) coordinates.

Parameters

- **settings** (*dict*) – Argument dict specifying which plate to define and with what (R,Z). See notes for more details.
- **rshift** (*float, optional*) – Translate the (R, Z) coordinates by a float value.
- **zshift** (*float, optional*) – Translate the (R, Z) coordinates by a float value.

Notes

Parameter `settings` is in the following form: `{plate_name: rz_data}`

where `plate_name` is a valid key string (see table below), and `rz_data` is an iterable structure with two iterable entries containing R and Z entries (see examples for usage).

Most users should use the Ingrid class method SetGeometry rather than interface directly with the IngridUtils method since SetGeometry calls this IngridUtils method after processing the user input.

Valid plate keys are as follows:

Plate	Accepted Keys (str)
Plate W1	`plate_W1`, `W1`
Plate E1	`plate_E1`, `E1`
Plate W2	`plate_W2`, `W2`
plate_E2	`plate_E2`, `E2`

Example

Defining target plate *plate_W1* with coordinates [(1,2), (2, 3), (3, 4)]

```
>>> MyIG = IngridUtils()
>>> r_entries = [1, 2, 3]
>>> z_entries = [2, 3, 4]
>>> rz_data = (r_entries, z_entries)
>>> MyIG.SetTargetPlate({'plate_W1': rz_data})
```

`WriteGridueDNL`(*gridue_settings: dict*, *fname: str = 'gridue'*) → bool
Write a gridue file for a double-null configuration.

Parameters

- **`gridue_settings`** (*`dict`*) – A dictionary containing grid data to be written to the gridue file.
- **`fname`** (*`str, optional`*) – The file name/path to save the gridue file to. Defaults to ‘gridue’.

Returns *True if file was written with no errors*

`WriteGridueSNL`(*gridue_settings: dict*, *fname: str = 'gridue'*) → bool
Write a gridue file for a single-null configuration.

Parameters

- **`gridue_settings`** (*`dict`*) – A dictionary containing grid data to be written to the gridue file.
- **`fname`** (*`str, optional`*) – The file name/path to save the gridue file to. Defaults to ‘gridue’.

Returns *True if file was written with no errors*

`get_config`() → str
Get the configuration obtained during analysis of x-points.

Returns A string identifying the configuration contained within LineTracing attribute *LineTracer*.

class `INGRID.utils.`**`TopologyUtils`**(*Ingrid_obj: object*, *config: str*)
Bases: `object`

The base class for all INGRID topologies.

Encapsulates key methods generating patch maps, visualizing data, generating grids, and exporting of grids.

These methods are to be interfaced by the child `ingrid.Ingrid` or can be used by advanced users of the code.

Variables

- **`parent`** (`Ingrid`) – Ingrid object the topology is being managed by.
- **`settings`** (*`dict`*) – Core settings dict of parent.
- **`CurrentListPatch`** (*`dict`*) – Lookup dict of Patches that have been refined (populated during Patch refinement).
- **`ConnexionMap`** (*`dict`*) – A mapping describing how Patch objects are connected to each other (see notes).
- **`CorrectDistortion`** (*`dict`*) – The settings to be used for correcting grid shearing.

- **config** (*str*) – Configuration of topology.
- **PlateData** (*dict*) – Parent target plate data dict.
- **PatchTagMap** (*dict*) – A dictionary containing the tag to patch name mappings for this topology.
- **LineTracer** (LineTracing) – Parent LineTracing instance.
- **PsiUNorm** (EfitData) – Parent PsiUNorm instance.
- **PsiNorm** (EfitData) – Parent PsiNorm instance.

CheckFunction(*expression: str*, *Verbose: bool = False*) → bool

Check if a str is in the correct format for method `get_func`

Parameters

- **expression** (*str*) – Expression to check.
- **Verbose** (*bool, optional*) – Print full output to terminal. Default to False

Returns *True if expression is valid. False otherwise.*

CheckPatches(*verbose: bool = False*) → None

Convenience method for calling the Patch class method `CheckPatch`.

Checks to make sure Patch objects stored in the TopologyUtils `patches` dictionary are monotonic in psi along strike geometry (if applicable).

Parameters **verbose** (*bool, optional*) – Print full output to terminal. Defaults to False.

GetBoundaryPoints(*AdjacentPatchInfo: tuple*) → list

Get the points along a boundary for a particular Patch.

Parameters **AdjacentPatchInfo** (*tuple*) – A 2-tuple containing the tag of the Patch to obtain boundary points *from* (str value), and a character indicating which boundary to access ('N', 'S', 'E', 'W').

Returns *A list containing the Points along the specified boundary for a Patch.*

GetDistortionCorrectionSettings() → dict

Get settings associated with the `CorrectDistortion` capability.

Returns A dictionary containing `CorrectDistortion` settings.

GetFunctions(*Patch:* INGRID.geometry.Patch, *Verbose: bool = False*) → tuple

Get the poloidal and radial transformations for a Patch.

Poloidal and radial transformations affect more than a single Patch in the index space. This method ensures that the same transformations are applied to dependent Patches (e.g. radial transformation T is applied to all Patch objects in the same radial level).

Parameters

- **Patch** (Patch) – The Patch to get the functions for.
- **Verbose** (*bool, optional*) – Print all output to terminal. Default to False.

Returns *2-tuple containing functions for radial and poloidal direction respectively.*

GetNpoints(*Patch:* INGRID.geometry.Patch) → tuple

Get the number of poloidal and radial grid cells to be generated for a Patch.

Because of index space dependence and formatting, this method ensures adjacent patches in the index space are matching in dimensions.

Parameters **`Patch`** (`Patch`) – Patch object to get the np and nr values for.

Returns *A 2-tuple containing the number of radial and poloidal cells to generate, respectively.*

`OrderPatches``()`

`RefreshSettings``()`

`SetPatchBoundaryPoints``(`*Patch: INGRID.geometry.Patch*, *verbose: bool = False*`)` → None

Set the Patch `BoundaryPoints` dict based off TopologyUtils `ConnexionMap`.

Parameters

- **`Patch`** (`Patch`) – The Patch to set the boundary points for.
- **`verbose`** (*`bool`*) – Print full output to terminal.

Notes

The `ConnexionMap` represents the layout of adjacent patches and will lookup what is adjacent to the Patch parameter being operated on.

`SetupPatchMatrix``()` → list

Instantiate the list representation of the Patch layout in index space.

Method `concat_grid` uses this structure when combining refined Patches into a global grid.

Returns *The 'PatchMatrix' list*

`concat_grid``(`*guard_cell_eps: float = 0.001*`)` → None

Concatenate a refined Patch map into a global grid.

This method take grid data and represents it into Fortran formatted arrays that will be written to gridue.

Adding of guard cells is done in this method as well.

Parameters **`guard_cell_eps`** (*`float, optional`*) – Determines the size of guard cell padding.

`construct_grid``(`*np_cells: int = 1*, *nr_cells: int = 1*, *Verbose: bool = False*, *ShowVertices: bool = False*, *RestartScratch: bool = False*, *ListPatches: str = 'all'*`)` → None

Construct a grid by refining a Patch map.

This method gathers transformation and dimension information to apply to each Patch. In addition, this applies any `CorrectDistortion` settings the user may want to apply.

Assumes a Patch map has been generated or that Patches have been loaded (equivalent).

Parameters

- **`np_cells`** (*`int, optional`*) – Number of poloidal cells to create in the local Patch grid. Defaults to 1.
- **`nr_cells`** (*`int, optional`*) – Number of radial cells to create in the local Patch grid. Defaults to 1.
- **`Verbose`** (*`bool, optional`*) – Print all output to terminal. Defaults to False
- **`ShowVertices`** (*`bool, optional`*) – Emphasize spline vertices on grid figure with bold markers. Defaults to False.
- **`RestartScratch`** (*`bool, optional`*) – Flag for repeating the Patch refinement process for an already refined Patch. Defaults to False.

- **ListPatches** (*str, optional*) – Specify which Patches to generate a grid for (see notes). Defaults to 'all'.

Examples

Parameter `ListPatches` can be used to specify which Patches to refine into a grid. The default value of *all* instructs TopologyUtils to refine the entire Patch map. To specify Patches, the user is to provide a list containing **names** of Patches to refine.

Refining only the outer-most psi boundary (SOL) for a lower single-null configuration:

```
>>> patch_names = ['A2', 'B2', 'C2', 'D2', 'E2', 'F2']
>>> MyTopologyUtils.construct_grid(np_cells=3, nr_cells=3, ListPatches=patch_
↪names)
```

Refining only the inboard and outboard psi boundary for an unbalanced double-null configuration:

```
>>> patch_names = ['A3', 'B3', 'C3', 'D3', 'E3', 'F3', 'G3', 'H3']
>>> MyTopologyUtils.construct_grid(np_cells=3, nr_cells=3, ListPatches=patch_
↪names)
```

Notes

The Patch refinement process is often order-dependent. This is to ensure alignment of grid with minimal editing.

Because of this, parameters such as `ListPatches` is suggested to be used by a user or developer who is sure of what they are doing.

get_config() → str

Return the configuration string stored in the TopologyUtils class

Returns *A string indicating the topology*

get_func(*func: str*) → function

Create a function from a string input.

Will be used to generate a poloidal or radial transformation function.

Parameters **func** (*str*) – An expression to generate a function from.

Returns *A function generated from the str input.*

Examples

When calling method `get_func` the user must provide a **string** with the following general format:

$$x, f(x)$$

That is, the dependent variable and expression to evaluate are separated with a comma character.

The *Sympy* library supports most expressions that can be generated with Python natively. See the *Sympy* documentation for advanced features.

Defining a function representing f(x) = x ^ 2:

```
>>> func = 'x, x ** 2'
>>> f = MyTopologyUtils.get_func(func)
>>> f(np.array([0, 1, 2, 3]))
array([0, 1, 4, 9])
```

Defining a function representing f(x) = exp(x)

```
>>> func = 'x, exp(x)'
>>> f = MyTopologyUtils.get_func(func)
>>> f(np.array([0, 1, 2, 3]))
array([ 1.        ,  2.71828183,  7.3890561 , 20.08553692])
```

grid_diagram(*fig: Optional[object] = None*, *ax: Optional[object] = None*) → None
Generates the grid diagram for a given configuration.

Parameters

- **fig** (*object, optional*) – Matplotlib figure to show the grid diagram on.
- **ax** (*object, optional*) – Matplotlib axes to plot the grid diagram on.

patch_diagram(*fig: Optional[object] = None*, *ax: Optional[object] = None*) → None
Generate the patch diagram for a given configuration.

Parameters

- **fig** (*object, optional*) – Matplotlib figure to show the Patch map on.
- **ax** (*object, optional*) – Matplotlib axes to plot the Patch map on.

3.4 geometry

The *geometry* module contains core classes that support the INGRID geometrical object hierarchy. This module also contains various helper functions that work in tandem with the *LineTracing* class to generate Patch maps and grids.

class INGRID.geometry.**Cell**(*lines*)
Bases: `object`

Define a Cell that resides within a grid.

Parameters **lines** (*array-like*) – A collection of 4 Line objects that define the borders of a Cell.

Variables

- **lines** (*array-like*) – The 4 Lines that create the border of the Cell.
- **vertices** (*dict*) – A lookup for accessing NW, NE, SE, SW, and CENTER spatial information.
- **p** (*list*) – A list of Point objects along the North and South border.

Notes

When accessing vertices, we have the following convention:

Location	Accepted Key (str)
NW Corner	`NW`
NE Corner	`NE`
SW Corner	`SW`
SE Corner	`SE`
Center	`CENTER`

`as_np``()` → numpy.ndarray

Get the ndarray representation of a Cell object

Returns *An ndarray representing a cell*

`plot_border``(`*color: str = 'red', ax: Optional[matplotlib.axes.Axes] = None*`)` → None

Plot the Cell.

Parameters

- **color** (`str, optional`) – Color of the Cell border. Defaults to 'red'
- **ax** (`matplotlib.axes.Axes, optional`) – The Axes instance to plot the Cell to.

`plot_center``(`*color='black', ax: Optional[matplotlib.axes.Axes] = None*`)` → None

Plot the center of a Cell.

Parameters

- **color** (`str, optional`) – The color of the marker. Defaults to 'black'
- **ax** (`matplotlib.axes.Axes, optional`) – The Axes instance to plot the Cell center to.

`class INGRID.geometry.`**`Line`**`(`*points: list*`)`

Bases: `object`

Define an arbitrary line/curve.

This is ordered collection of Point objects can later be used to define a Patch object.

Parameters **points** (`list`) – The Point objects that define the Line.

Variables

- **p** (`list`) – The list of Point objects that define this Line.
- **xval** (`list`) – A list consisting the x-coordinates for each Point.
- **yval** (`list`) – A list consisting the y-coordinates for each Point.

`GetAngle``(`*Line*`)`

Return the angle between two lines in degree (between 0 and 180 degrees) :param Line: DESCRIPTION. :type Line: TYPE

Returns None.

`Norm``()`

Return norm of the lines :returns: None.

`RemoveDuplicatePoints``()`

Remove any duplicate points from list of points

as_np() → numpy.ndarray

Get the calling Line object represented as an ndarray.

Returns *An ndarray representation of the Line.*

Notes

Format of ndarray is of shape (2, n), with n being the number of Point objects in the Line.

The first entry of the ndarray is the `xval` attribute. The second entry of the ndarray is the `yval` attribute.

This method is used to encode the *patch_data* file.

copy() → *INGRID.geometry.Line*

Create a copy of this Line object.

Returns *A new Line instance.*

fluff(*num: int = 1000*, *verbose: bool = False*) → tuple

Obtain linspaced copies of the `xval` and `yval` attributes.

Parameters

- **num** (*int, optional*) – Number of entries to include between **each** segment within the Line. Defaults to 100.
- **verbose** (*bool, optional*) – Print full output to terminal. Defaults to False

Returns A 2-tuple consisting of 'fluffed' `xval` and `yval`.

fluff_copy(*num: int = 5*) → *INGRID.geometry.Line*

Create a 'fluffed' copy of this Line.

Calls the method `fluff` internally.

Parameters **num** (*int, optional*) – Number of entries to include between **each** segment within the Line copy.

Returns *A 'fluffed' copy of the calling Line object.*

plot(*color: str = '#1f77b4'*, *label: str = ''*, *ax: Optional[matplotlib.axes.Axes] = None*, *linewidth: float = 1.0*) → matplotlib.axes.Axes

Plot the Line.

Parameters

- **color** (*str, optional*) – Defaults to a light blue.
- **label** (*str, optional*) – A label to plot with. Defaults to None.
- **ax** (*matplotlib.axes.Axes, optional*) – The Axes instance to plot the Line to.
- **linewidth** (*float, optional*) – The linewidth to plot with.

Returns *The matplotlib.axes.Axes instance plotted on.*

points() → list

Get a list of all coordinates within the Line object.

Returns *A list of tuples representing (x, y) coordinates of the Line.*

print_points() → None

Prints each point in the line to the terminal.

reverse_copy() → *INGRID.geometry.Line*

Create a copy of this Line in reversed order.

Returns *A new Line instance*

split(*split_point*, *add_split_point=False*) → tuple

Split a line object into two line objects at a particular point.

Returns two Line objects Segment A and Segment B (corresponding to both subsets of Points) The split_point is always included in Segment B.

Parameters

- **split_point** (Point) – Point that determines splitting location.
- **add_split_point** (*bool*) – Append the split point to Segment A while still including the split point in Segment B.

Returns *A tuple with Line objects representing Segment A and Segment B.*

class INGRID.geometry.**Patch**(*lines: array - like*, *patch_name: str = ''*, *PatchTagMap: dict = None*, *plate_patch: bool = False*, *plate_location: str = None*, *color: str = 'blue'*)

Bases: object

Define a Patch representing a portion of the tokamak domain.

Each Patch can be refined into a subgrid that can then create a global grid.

Parameters

- **lines** (*array-like*) – The four Line objects defining the boundary of this Patch (N, E, S, W).
- **patch_name** –

AdjustBorder(*face*, *patch*)

CheckPatch(*grid*, *verbose=False*)

RemoveDuplicatePoints()

adjust_corner(*point*, *corner*)

as_np()

cell_grid_as_np()

fill(*color='lightsalmon'*, *ax=None*, *alpha=1.0*)

Shades in the patch with a given color

Parameters **color** (*str, optional*) – Defaults to a light salmon.

get_settings()

get_tag()

make_subgrid(*grid*, *np_cells=2*, *nr_cells=2*, *_poloidal_f=<function Patch.<lambda>>*, *_radial_f=<function Patch.<lambda>>*, *verbose=False*, *visual=False*, *ShowVertices=False*)

Generate a refined grid within a patch. This ‘refined-grid’ within a Patch is a collection of num x num Cell objects

Parameters

- **grid** (Ingrid) – To be used for obtaining Efit data and all other class information.
- **num** (*int, optional*) – Number to be used to generate num x num cells within our Patch.

plot_border(*color='red'*, *ax=None*)

Draw solid borders around the patch.

Parameters **color** (*str, optional*) – Defaults to red.

`plot_subgrid`(*fig=None*, *ax=None*, *color='blue'*)

class `INGRID.geometry.Point`(**pts*)

Bases: `object`

Define a Point.

Can be used to later define Line objects.

Parameters **pts** (*array-like*) – Accepts either two values x, y as floats, or a single tuple/list value (x, y).

Variables

- **x** (*float*) – x coordinate of the point
- **y** (*float*) – y coordinate of the point
- **coor** (*tuple*) – x and y coordinates together as a tuple

`as_np`() → numpy.ndarray

Return the Point object as a numpy ndarray.

Returns *An ndarray representation of the Point object.*

`plot`(*ax: Optional[matplotlib.axes.Axes] = None*) → None

Plot the Point.

Parameters **ax** (*matplotlib.axes.Axes, optional*) – The Axes instance to plot to. Default is None and calls function *matplotlib.pyplot.gca*.

`psi`(*grid: EfitData*, *tag: str = 'v'*) → float

Get the psi value of this Point from an EfitData instance.

Parameters

- **grid** (EfitData) – The grid upon which the value of psi is to be calculated on.
- **tag** (*str, optional*) – Char to specify the type of psi derivative. Defaults 'v' (no derivative).

Returns *The psi value at the Point.*

class `INGRID.geometry.Vector`(*xy: array - like*, *origin: array - like*)

Bases: `object`

Defines a vector from a nontrivial origin.

Parameters

- **xy** (*array-like*) – Location of the vector. It if of the form (x, y).
- **origin** (*array-like*) – Location of the origin. This is to adjust for not being at the origin of the axes. Of the form (x, y).

Variables

- **x** (*float*) – x-coordinate
- **y** (*float*) – y-coordinate
- **xorigin** (*float*) – x-coordinate of vector origin
- **yorigin** (*float*) – y-coordinate of vector origin
- **xnorm** (*float*) – x relative to origin
- **ynorm** (*float*) – y relative to origin

- **quadrant** (*int*) – Quadrant vector resides in

arr() → numpy.ndarray
Return the vector object as an array.

Returns *The vector as an numpy ndarray.*

mag() → float
Return the L2 norm of the vector.

Returns *Vector norm.*

INGRID.geometry.**CorrectDistortion**(*u*, *Pt*, *Pt1*, *Pt2*, *spl*, *ThetaMin*, *ThetaMax*, *umin*, *umax*, *Resolution*, *visual*, *Tag*, *MinTol=1.02*, *MaxTol=0.98*, *Verbose=False*)

INGRID.geometry.**UnfoldLabel**(*Dic: dict*, *Name: str*) → str
Unfold Patch label (e.g. "C1" -> "Inner Core Top")

Parameters

- **Dic** (*dict*) – Dictionnary containing description of acronym characters
- **Name** (*str*) – patch label

Returns *str* – Unfolded patch label.

INGRID.geometry.**angle_between**(*u*, *v*, *origin*, *relative=False*)
Compute angle in radians between vectors u and v

INGRID.geometry.**calc_mid_point**(*v1*, *v2*)
Calculates the bisection of two vectors of equal length, and returns the point on the circle at that angle.

Parameters

- **v1** (geometry.Vector) – v1 must be furthest right in a counter clockwise direction.
- **v2** (geometry.Vector) – Vector on the left.

Returns *tuple* – The point at the bisection of two vectors.

INGRID.geometry.**find_split_index**(*split_point:* INGRID.geometry.Point, *line:* INGRID.geometry.Line) → tuple
Determine which index a Point would best split a Line.

This method is useful for modifying Line objects during line tracing (searching for intersection of two line objects and trimming excess)

Parameters

- **split_point** (Point) – The candidate Point to find the split index with respect to.
- **line** (Line) – The Line to search for a split index within.

Returns *A 2-tuple containing the split-index and a boolean flag indicating* – whether the split_point was contained within the Line

Notes

Should no appropriate split index be found, the method will return a None value in place of an integer index.

The second entry of the tuple return value would be a value of True if the *split_point* parameter was used to define the *line* parameter.

`INGRID.geometry.`**`intersect`**`(`*line1*, *line2*, *verbose=False*`)`
Finds the intersection of two line segments

Parameters

- **`line1`** (*`array-like`*) –
- **`line2`** (*`array-like`*) – Both lines of the form A = ((x, y), (x, y)).

Returns *tuple* – Coordinates of the intersection.

`INGRID.geometry.`**`is_between`**`(`*end_u: array - like*, *split_v: array - like*`)` → bool

`INGRID.geometry.`**`limiter_split`**`(`*start*, *end*, *limiter*`)`

`INGRID.geometry.`**`non_decreasing`**`(`*L: array - like*`)` → bool
Determine if non-decreasing.

Parameters **`L`** (*`array-like`*) – Values to test.

Returns *True if non-decreasing and False otherwise*

`INGRID.geometry.`**`non_increasing`**`(`*L: array - like*`)` → bool
Determine if non-increasing.

Parameters **`L`** (*`array-like`*) – Values to test.

Returns *True if non-increasing and False otherwise*

`INGRID.geometry.`**`orientation_between`**`(`*u*, *v*, *origin*`)`
Compute angle in radians between vectors u and v

`INGRID.geometry.`**`reorder_limiter`**`(`*new_start*, *limiter*`)`

`INGRID.geometry.`**`rotate`**`(`*vec*, *theta*, *origin*`)`

`INGRID.geometry.`**`rotmatrix`**`(`*theta: float*`)` → numpy.ndarray
Construct a rotation matrix

Parameters **`theta`** (*`float`*) – Angle in radians.

Returns *An ndarray with shape (2, 2) representing a 2D rotation matrix.*

`INGRID.geometry.`**`segment_intersect`**`(`*line1*, *line2*, *verbose=False*`)`
Finds the intersection of two FINITE line segments. :param line1: :type line1: array-like :param line2: Both lines of the form line1 = (P1, P2), line2 = (P3, P4) :type line2: array-like

Returns *bool, tuple* – True/False of whether the segments intersect Coordinates of the intersection

`INGRID.geometry.`**`strictly_decreasing`**`(`*L: array - like*`)` → bool
Determine if strictly decreasing.

Parameters **`L`** (*`array-like`*) – Values to test.

Returns *True if strictly decreasing and False otherwise*

`INGRID.geometry.`**`strictly_increasing`**`(`*L: array - like*`)` → bool
Determine if strictly increasing.

Parameters **`L`** (*`array-like`*) – Values to test.

Returns *True if strictly increasing and False otherwise*

`INGRID.geometry.`**`test2points`**`(`*p1*, *p2*, *line*`)`
Check if two points are on opposite sides of a given line.

Parameters

- **`p1`** (`tuple`) – First point, (x, y)
- **`p2`** (`tuple`) – Second point, (x, y)
- **`line`** (`array-like`) – The line is comprised of two points ((x, y), (x, y)).

Returns *tuple* – Returns two numbers, if the signs are different the points are on opposite sides of the line.

`INGRID.geometry.`**`trim_geometry`**`(`*geoline*, *start*, *end*`)`

`INGRID.geometry.`**`unit_vector`**`(`*v*`)`
Returns unit vector

`INGRID.geometry.`**`which_increasing`**`(`*L: array - like*`)` → list
Determine increasing values.

Parameters **`L`** (`array-like`) – Values to test.

Returns *A list of 2-tuples containing index and increasing element*

`INGRID.geometry.`**`which_non_increasing`**`(`*L: array - like*`)` → list
Determine non-increasing values.

Parameters **`L`** (`array-like`) – Values to test.

Returns *A list of 2-tuples containing index and non-increasing element.*

3.5 line_tracing

`exception` `INGRID.line_tracing.`**`RegionEntered`**`(`*message*, *region*`)`
Bases: `Exception`

`class` `INGRID.line_tracing.`**`LineTracing`**`(`*grid*, *settings*, *eps=1e-06*, *tol=5e-05*, *first_step=1e-05*, *numPoints=25*, *dt=0.01*, *option='xpt_circ'*, *direction='cw'*`)`
Bases: `object`

This class traces the polodal and radial lines of a given psi function based of the points where the user clicks.

Parameters

- **`grid`** (`EfitData.EfitData`) – The grid object upon which the lines will be drawn.
- **`settings`** (`dict`) – YAML file containing all INGRID parameters.
- **`eps`** (`float, optional`) – Short for epsilon. Specifies the size of the circle drawn around the zero point.
- **`tol`** (`float, optional`) – Short for tolerance. Specifies how close to the final point the line must get before converging. Also defines a circle.
- **`numPoints`** (`int`) – Number of points in the circle of radius eps.
- **`dt`** (`float, optional`) – Specify the size of each line segment that is traced by scipy.integrate.solve_ivp.

- **option** (*str, optional*) – 'theta' draws the poloidal line where the user clicks. 'rho' draws the radial line where the user clicked. 'xpt_circ': uses the root finder to find the root closest to where the user clicked. Then finds the points around that circle a distance epsilon away.
- **direction** (*str, optional*) – 'cw' or 'ccw'. Specifies clockwise or counterclockwise line tracing.

DNL_find_NSEW(*xpt1*, *xpt2*, *magx*, *visual=False*)
Find NSEW based off primary x-point and magnetic axis,

Parameters

- **xpt** (*array-like*) – R, Z coordinate of the primary x-point.
- **mag** (*array-like*) – R, Z coordinate of the magnetic axis.

Notes

LineTracer_psi will contain NSEW information post call.

PsiCostFunc(*xy*)

SNL_find_NSEW(*xpt*, *magx*, *visual=False*)
Find NSEW based off primary x-point and magnetic axis,

Parameters

- **xpt** (*array-like*) – R, Z coordinate of the primary x-point.
- **mag** (*array-like*) – R, Z coordinate of the magnetic axis.

Notes

self.LineTracer_psi will contain NSEW information post call.

analyze_saddle(*xpt*, *xpt_ID*)
Finds theta values to be tested for N and S directions

disconnect()
Turns off the click functionality

draw_line(*rz_start*, *rz_end=None*, *color='purple'*, *option=None*, *direction=None*, *show_plot=False*, *text=False*, *dynamic_step=None*, *debug=False*, *Verbose=False*)
Uses scipy.integrate.solve_ivp to trace poloidal or radial lines. Uses the LSODA method to solve the differential equations. Three options for termination criteria, specified by rz_end.

Parameters

- **rz_start** (*array-like or* geometry.Point) – Starting location for line tracing.
- **rz_end** (*dict, optional*) – Defaults to be rz_start. This is how we specify the termination critera. i.e. {'point': Point}, {'line': Line}, {'psi': Psi} Points can be a geometry.Point, or array-like i.e. (x, y) Lines can be a geometry.Line, or array-like i.e. ((x, y), (x, y)) Psi must be a scalar, i.e. 1.1, and specifies the level of psi to stop on.
- **color** (*str, optional*) – Specifies the color of the produced grid lines.
- **option** (*str, optional*) – Change which differential equation is used in the line tracing proccess. 'theta', 'rho'

- **direction** (*str*) – determines if the function plots clockwise (cw) or counterclockwise (ccw). default is None.
- **show_plot** (*bool, optional*) – Show the user real-time tracing and the line tracer works.
- **text** (*bool, optional*) – Prints convergence method, number of iterations, and time taken to the terminal window.

Returns **line** (*geometry.Line*) – Curved line consisting of the start and end points of each segment calculated by solve_ivp. Does not store the intermediate points.

flip_NSEW_lookup(*xpt_ID*)

map_xpt(*xpt*, *magx*, *xpt_ID='xpt1'*, *visual=False*, *verbose=False*)

rotate_NSEW_lookup(*xpt_ID*, *turns=2*)

3.6 interpol

Module containing EfitData class for handling all interpolation related computations.

class INGRID.interpol.**EfitData**(*rmin=0.0*, *rmax=1.0*, *nr=10*, *zmin=0.0*, *zmax=2.0*, *nz=20*, *rcenter=1.6955*, *bcenter=- 2.1094041*, *rlimiter=None*, *zlimiter=None*, *rmagx=0.0*, *zmagx=0.0*, *name='unnamed'*, *parent=None*)

Bases: `object`

Structure to store the rectangular grid of psi data. It uses cylindrical coordinates, where R and Z are similar to the cartesian x and y. The phi components goes away due to the symmetry of a tokamak.

Parameters

- **rmin** (*float, optional*) – left boundary of the grid
- **rmax** (*float, optional*) – right boundary
- **nr** (*int, optional*) – number of grid points in the R direction
- **zmin** (*float, optional*) – bottom boundary for the grid
- **zmax** (*float, optional*) – top boundary
- **nz** (*int, optional*) – number of grid points in the Z direction
- **name** (*str, optional*) – Specify the title of the figure the data will be plotted on.

Gradient(*xy: tuple*) → numpy.ndarray

Combines the first partial derivatives to solve the system for maximum, minimum, and saddle locations.

Parameters **xy** (*array-like*) – Contains x and y. Ex: xy = (x0, y0).

Returns **F** (*array*) – Vector function to be used in find root.

Hessian(*xy: tuple*) → numpy.ndarray

Compute the Hessian at a point.

Parameters **xy** (*array-like*) – Contains x and y. Ex: xy = (x0, y0).

Returns **H** (*array*) – Numpy array of shape (2, 2) representing the Hessian at xy.

PlotLevel(*level: float = 1.0*, *color: str = 'red'*, *label: str = ''*, *linestyles: str = 'solid'*, *refined: bool = True*, *refine_factor: int = 10*) → None

Plot a psi level and provide it a label.

This function is useful for management of psi boundaries such as 'psi_pf', 'psi_core', etc and ensuring the contour will be properly replotted (no duplicate of same label).

Parameters

- **level** (*float, optional*) – Psi level to plot. Default to 1.0 (separatrix of normalized psi)
- **color** (*str, optional*) – Color to pass to matplotlib contour function
- **label** (*str, optional*) – Label to associate with the psi level
- **linestyles** (*str, optional*) – Line style to pass to matplotlib contour function
- **refined** (*bool, optional*) – Plot level with hi-resolution cubic spline representation
- **refine_factor** (*int, optional*) – Refinement factor for to be passed to SciPy zoom method

PsiFunction(*xy*)

clear_plot()

get_psi(*r0*, *z0*, *tag='v'*)

find grid cell encompassing (r0,z0) note: grid is the crude grid. Uses Bicubic Interpolation to calculate the exact value at the point. Useful for finding information inbetween grid points.

Parameters

- **r0** (*float*) – R coordinate of the point of interest
- **z0** (*float*) – Z coordinate of same point.
- **tag** (*str, optional*) – tag is the type of derivative we want: v, vr, vz, vrz if nothing is provided, it assumes no derivative (v).

Returns *float* – Value of psi or its derviative at the coordinate specified.

init_bivariate_spline(*r: numpy.ndarray*, *z: numpy.ndarray*, *v: numpy.ndarray*) → None

Initialize scipy.interpolate.RectBivariateSpline object for Bicubic interpolation.

Sets class member v to crude EFIT grid.

Parameters

- **r** (*array-like*) – 1-D array of r coordinates in strictly ascending order.
- **z** (*array-like*) – 1-D array of z coordinates in strictly ascending order.
- **v** (*array-like*) – 2-D array of EFIT data with shape (r.shape, z.shape)

plot_data(*nlevs: int = 30*, *interactive: bool = True*, *fig: Optional[object] = None*, *ax: Optional[object] = None*, *view_mode: str = 'filled'*, *refined: bool = True*, *refine_factor: int = 10*)

Plot the EFIT data.

Visualizes eqdsk file with either contour lines or filled contours.

Parameters

- **nlev** (*int, optional*) – number of levels we want to be plotted
- **interactive** (*bool, optional*) – Set matplotlib interactive mode on or off
- **fig** (*object, optional*) – Matplotlib figure handle
- **ax** (*object, optional*) – Matplotlib axes handle

- **view_mode** (*str, optional*) – Represent EFIT data with standard contour lines or filled contour lines. String value of 'filled' enables filled contours, whereas 'lines' omits filling of contours.
- **refined** (*bool, optional*) – Plot level with hi-resolution cubic spline representation
- **refine_factor** (*int, optional*) – Refinement factor for to be passed to SciPy zoom method

plot_levels(*level=1.0*, *color='red'*)

This function is useful if you need to quickly see where a particular line of constant psi is. It in't able to store points of intersection, and cannot be generalized. If you need just a segment of psi, use the draw_lines method in the line tracing class.

Parameters

- **level** (*float, optional*) – Value of psi you wish to see
- **color** (*str, optional*) – color of the line.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX