Simulation of CXRS spectra using Raysect and CHERAB frameworks

Aleksei Shabashov
a.shabashov@iterrf.ru

Contents

Appendices

A For Users

A.1 Installation

Clone this repository then go to its root folder:

```
git clone ssh://git@git.iter.org/diag/cxrs.git
cd cxrs
```

First of all you have to load all required modules:

```
source env.sh
```

This will purge all loaded modules and load ones that necessary to install and use `cxrs`.
To install `cxrs` on your computer run

```
pip install --user .
```

while in the the `cxrs` root directory.

Note: you can omit `python -m` part later by adding `$HOME/.local/bin` to your `PATH`. To do so, locate file `.bashrc` in your home directory and add the following line at the end of the file:

```
export PATH="$PATH:/home/ITER/<username>/.local/bin"
```

where `<username>` is your username at ITER GPC. You can look at it by executing next command via command line: `echo $USER`.

A.2 Preparations for the first run

Step 1: Create environment file

In order to create environment file, run

```
python -m cxrs create-env
```

This will create the default environment file, that loads all needed modules. To do so, run

```
source env.sh
```

It will purge all loaded modules and will load modules necessary to use `cxrs`.

Note: you have to do this for every new session on ITER GPC.

Step 2: Create configuration file

`cxrs` behavior controlled by configuration file which is necessary to run the code.

To create default configuration file, run

```
python -m cxrs create-config
```

This will create a configuration file with a name `config.xml` in current directory.

Step 3: Populate local atomic database

`cxrs` uses a local atomic database. To create one, run

```
python -m cxrs populate
```

Atomic data will be copied to the `$HOME/.cherab` directory.

A.3 Usage

A.3.1 Performing a simulation

Main part of `cxrs` is simulation routine, to use it for pulse with `<shot>` shot number and `<run>` run number for time slice `<time>` use:

```
python -m cxrs simulate -s <shot> -r <run> -t <time> -c <config>
```

where `<config>` is a path to configuration file.

It will perform a simulation and will store result in newly created folder `cxrs_output`.

Note: `time` can be omitted. In that case time slice in the middle of the time range will be used. This is particularly useful for time slices with a lot of digits after after decimal separator. For example :

```
python -m cxrs simulate -s 134000 -r 30 -c config.xml
```

For additional information use help:

```
python -m cxrs simulate --help
```

A.3.2 Print plasma profiles

`cxrs info` subprogram can be used to inspect different distributions of plasma and DNB parameters. To use it, run

```
python -m cxrs info -s <shot> -r <run>
```

It will produce variety of plots and tables and place it in `cxrs_output` folder. *Note:* 1D profiles represent values at the DNB axis.

A.3.3 Print plasma composition

You can quickly look at plasma composition for requested pulse by using

```
python -m cxrs composition -s <shot> -r <run>
```

Omit arguments to print composition for all pulses available by `scenario_summary` program.

A.3.4 Configuration

`cxrs` accepts user's configurations as xml-file.

To change option's value, change value attribute in double quotes corresponding to the option.

For more information on the meaning of certain parameter, look at its description.

Note: type attribute is actually important. For string values use `str`, for floating-point values use `float` and `int` for integers.

Warning: Do not change tags, like `<user_options>` or others, this will stop the program from working.

A.3.5 Emission parameters

Section `emission_parameters` controls which emission types are used during a simulation. Each of them can be turned on or off separately.

A.3.6 Plasma, beam, fibre, optics, camera, scanner and spectrometer parameters

Each of those sections controls appropriate aspect of simulation. For information on each parameter look at its description in configuration file.

A.3.7 Emission lines

List of emission lines suited for observations is placed in `<emission_parameters>` section:

```
<emission_lines description="CXRS elements to observe.">
    <HI>
        <name
            description="Name of the element."
            unit="n.a."
            type="str"
            value="hydrogen"/>
        <charge
            description="Charge of the ion."
            unit="n.a."
            type="int"
            value="0"/>
        <transition_levels
            description="Transition levels [upper, lower]."
            unit="n.a."
            type="int"
            value="[3, 2]"/>
    </HI>
    ... more lines omitted
</emission_lines>
```

You can change and add new entries in this section following given template. Added emission lines will be simulated if atomic data exists for them.

A.3.8 DNB

DNB section of the configuration file contains parameters for Diagnostic Neutral Beam. Structure of this section replicates structure of nbi IDS.

A.3.9 Wavelength ranges and geometry

These two sections contain all information for CXRS diagnostics.

B For Developers

This section is aimed to help future developers in their work on `cxrs`. Most of the functions implemented in this project contain docstrings written in "classic" Python style using **reStructuredText** syntax. Each docstring contains a short description of a function, list of arguments and their types and list of exceptions which can be risen during execution of the function, but description for some of the arguments can be missing.

Also this section tries to be a proper documentation for a software project. But, really, it is more of a reasoning for why *this thing is done like that and not like this*.

B.1 Project Structure

`cxrs` project is divided in several modules, each in its respective folder:

```
/
└── cxrs/
    ├── imas/
    │   ├── __init__.py
    │   ├── charge_exchange.py
    │   ├── core_profiles.py
    │   ├── edge_profiles.py
    │   ├── equilibrium.py
    │   ├── find_nearest.py
    │   ├── ids.py
    │   └── nbi.py
    ├── machine/
    │   ├── portplugs/
    │   ├── simple/
    │   ├── __init__.py
    │   └── pfc_mesh.py
    ├── model/
    │   ├── __init__.pxd
    │   ├── __init__.py
    │   ├── passive.pxd
    │   └── passive.pyx
    ├── observers/
    │   ├── __init__.py
    │   ├── base.py
    │   ├── camera.py
    │   ├── fibre_bundle.py
    │   ├── fibres.py
    │   ├── optics.py
    │   ├── scanner.py
    │   ├── sightlines.py
    │   ├── spectrometer.py
    │   └── total_radiance.py
    └── populate/
```

```
    ├── __init__.py
    ├── create.py
    └── openadas.py
├── subprograms/
    ├── __init__.py
    ├── composition.py
    ├── idslist.py
    ├── info.py
    ├── local_copy.py
    ├── populate.py
    ├── read_ids.py
    ├── search.py
    ├── simulate.py
    └── write_ids.py
├── utility/
    ├── __init__.py
    ├── info/
        ├── __init__.py
        ├── beam.py
        ├── equilibrium.py
        ├── passive.py
        ├── plasma.py
        └── profiles.py
    ├── annotation.py
    ├── data.py
    ├── emission.py
    ├── fit.py
    ├── math.py
    ├── timer.py
    └── xml.py
├── __init__.py
├── __main__.py
├── create_beam.py
├── env.sh
└── matplotlibrc
├── tests/
├── .gitignore
├── config.xml
└── env.sh
```

```
├── MANIFEST.in
├── matplotlibrc
├── README.md
├── requirements.txt
├── setup.cfg
└── setup.py
```

B.2 subprograms module – Command Line Interface

`cxrs` uses CLI with git-like structure with subprograms. They are defined in `cxrs/subprograms` module.

Script controlling CLI is `cxrs/__main__.py`. It defines `CLI` class and all subprograms as its methods. Each method calls `main` function from respective module and passes arguments from command line to it.

B.2.1 composition

`main` function is called when CLI is used:

```
python -m cxrs composition -s 134000 -r 30 -f 134000_30_composition.txt
```

```
main(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
    filename="134000_30_composition.txt"
)
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number:
`filename` path to output file. If not specified, output will be printed to stdout.

Note: if `shot` and `run` are not specified program will print compositions for all available pulses.

`composition_core` function prints core plasma composition for requested pulse.

```
composition_core(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3"
)
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number.

`composition_edge` function prints core plasma composition for requested pulse.

```
composition_edge(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3"
)
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number.

`scenario_summary` function returns an output of `scenario_summary -c shot,run` as a string.

```
scenario_summary()
```

B.2.2 `info`

`main` function is called when CLI is used:
`python -m cxrs info -s 134000 -r 30 -c config.xml`. It reads IMAS data for requested pulse, builds appropriate models: EFITEquilibrium, Plasma, Beam and produces useful information in form of plots and text tables. It allows to quickly look at models before performing computational-heavy observation tasks. Functions for creating plots and tables as placed in `cxrs/utility/info` folder in respective files.

```
main(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
    filename="config.xml"
)
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number:
`filename` path to configuration file.

B.2.3 `populate`

`main` function is called when CLI is used:
`python -m cxrs populate`
It calls `populate_more` function from `cxrs/populate/create.py` with single argument `adas_path="/work/projects/adas/adas/"`. This populates local CHERAB's atomic data repository at `/home/$USER/.cherab` folder with data defined in the body of `populate_more` function and makes `cxrs` use ADAS rate files stored in `/work/projects/adas/adas`.

B.2.4 `read_ids`

Contain functions that are used to read necessary data from charge_exchange and nbi IDSs. Designed for debug purposes.

`cxs` function is called when CLI is used:
`python -m cxrs read_ids -s 1 -r 1 -u shabasa -d test --cxs -o 0`
Prints all information related to diagnostic geometry stored in `charge_exchange` IDS for requested pulse.

```
cxs(
    shot=1,
    run=1,
    user="shabasa",
    database="test",
    version="3",
```

```
        occurrence=0
    )
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number;
`occurrence` IDS occurrence ID.

`nbi` function is called when CLI is used:
`python -m cxrs read_ids -s 1 -r 1 -u shabasa -d test --nbi -o 0`
Prints all information related to DNB geometry and parameters stored in `nbi` IDS for requested pulse.

```
    nbi(
        shot=1,
        run=1,
        user="shabasa",
        database="test",
        version="3",
        occurrence=0
    )
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number;
`occurrence` IDS occurrence ID.

B.2.5 `write_ids`

Contain functions that are used to write `charge_exchange` and `nbi` data from configuration file to appropriate IDSs.

`cxs` function is called when CLI is used:
`python -m cxrs write_ids -c config.xml -s 1 -r 1 -u shabasa -d test --cxs -o 0`
Creates a `charge_exchange` IDS instance in a local database with contents of the configuration file.

```
    cxs(
        shot=1,
        run=1,
        user="shabasa",
        database="test",
        version="3",
        config="config.xml",
        occurrence=0
    )
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number;
`config` path to configuration file;
`occurrence` IDS occurrence ID.

`nbi` function is called when CLI is used:
`python -m cxrs write_ids -c config.xml -s 1 -r 1 -u shabasa -d test --nbi -o 0`
Creates a `nbi` IDS instance in a local database with contents of the configuration file.

```
    nbi(
        shot=1,
        run=1,
        user="shabasa",
        database="test",
        version="3",
        config="config.xml",
        occurrence=0
    )
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number;
`config` path to configuration file;
`occurrence` IDS occurrence ID.

B.2.6 `search`

Search for pulses that contain requested element by its symbol.

`main` function is called when CLI is used:
`python -m cxrs search C`
Prints list of pulses with their shot and run numbers. Uses `scenario_summary` program.

```
main(
    symbol="C",
    verbose=False,
    clean=False,
)
```

`symbol` symbol of the requested element;
`verbose` print additional text;
`clean` delete temporary file produced by `scenario_summary` program.

B.2.7 `simulate`

`main` function is called when CLI is used:
`python -m cxrs simulate -s 134000 -r 30 -t -1 -c config.xml`
Main routine for CXRS spectra simulation. Results are stored in `cxrs_output/shot_<shot>_run_<run>/tim`
directory.

```
main(
    shot=134000,
    run=30,
    time=-1.0,
    user=134000,
    database=134000,
    version=134000,
    config="config.xml",
)
```

`shot` IMAS database shot ID;
`run` IMAS database run ID;
`time` requested time;
`user` IMAS user ID;
`database` IMAS database ID;
`version` IMAS major version number;
`config` path to configuration file.

Note: `-t -1` can be used to choose time slice in the middle of the time range or it case when IDS contains only one time slice.

`print_parameters` function is used to print simulation parameters defined in the configuration file to the stdout.

```
main(
    user_options,
    plasma_parameters,
    beam_parameters,
    emission_parameters
)
```

`user_options` parameters from ¡user_options¿ section of the configuration file;
`plasma_parameters` parameters from ¡plasma_paragrameters¿ section of the configuration file;
`beam_parameters` parameters from ¡beam_parameters¿ section of the configuration file;
`emission_parameters` parameters from ¡emission_parameters¿ section of the configuration file.

B.3 `imas` module – Interaction with IMAS database

Module responsible for reading data from IMAS and creating appropriate plasma and DNB models is stored in `cxrs/imas` directory.

B.3.1 Supplementary Functions

`ids_get` File `ids.py` contains only one function `ids_get` which is used to check and load requested IDS. For example

```
charge_exchange_ids = ids_get(
    name="charge_exchange",
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
    occurrence=0
)
```

`name` is a name of requested IDS,
`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,

`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,
`occurrence` is an IDS occurrence number. Each IDS can store several `occurrences` (refer to the description of an IDS). For example in `charge_exchange` IDS occurrences used to store data related to different diagnostics.

`find_nearest` File `find_nearest.py` contains only one function `find_nearest`. It is used for locating an index of the nearest time slice to the time requested by user.

Example:

```
idx = find_nearest(
    array=time_slices,
    value=260,
)
```

`array` is an input array,
`value` is a value to search for.

B.3.2 `equilibrium` IDS

File `equilibrium.py` contains `EquilibriumIDS` class which is used to read data from `equilibrium` IDS and create CHERAB's `EFITEquilibrium` object via `time` method. It is later used to build core plasma model.

Load an `equilibrium` IDS:

```
equilibrium_ids = EquilibriumIDS(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
)
```

`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,
`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,

Create an `EFITEquilibrium` object:

```
equilibrium = equilibrium.time(time=-1)
```

`time` is a requested time. Here `-1` for time is used to acquire a time slice in the middle of time range.

Other methods:

`ids()` returns the `equilibrium` IDS object;

`psi_1d()` returns one-dimensional Ψ profile stored in `ids.time_slice[i].profiles_1d.psi`. It is implemented in case if `core_profiles` (section B.3.3) IDS does not contain its own profile.

B.3.3 `core_profiles` IDS

File `core_profiles.py` contains `CoreProfilesIDS` class which is used to read data from `core_profiles` IDS and create CHERAB's `Plasma` object via `create_plasma` method. It is poses as core plasma model.

Load a `core_profiles` IDS:

```
core_profiles_ids = CoreProfilesIDS(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
)
```

`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,
`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,

Create a `Plasma` object:

```
core_plasma = core_profiles_ids.create_plasma(
    equilibrium=equilibrium,
    psi_1d=equilibrium.psi_1d(),
    integration_step=0.001,
    integration_samples=5,
    parent=world,
    transform=None,
    name="Core Plasma"
)
```

`equilibrium` is an EFITEquilibrium object (section B.3.2);

`psi_1d` is one-dimensional Ψ profile (section B.3.2). It is used if one stored in `core_profiles` IDS is missing;

`integration_step` is step of volumetric integration in meters;

`integration_samples` is a number of integration samples;

`parent` is a parent node;

`transform` is transformation matrix;

`name` is a name of this plasma.

Note: for more information on `integration_step` and `integration_samples` refer to CHERAB's documentation on `Plasma`. For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

Note that `create_plasma` method does not require time value since it uses one that stored in `equilibrium`.

One required argument of `create_plasma` is `equilibrium`. It provides Ψ_{norm} distribution which is used to map density, temperature and bulk velocity distributions of plasma. Functions `distribution_density`, `distribution_temperature` and `distribution_velocity` are doing exactly that. *Note:* `distribution_temperature` tries to use average ion temperature or electron temperature if species' own temperature profile is absent in the IDS.

`detect_species` Function `detect_species` is used to recognize ion and neutral species from label given in IDS. Since there was no convention on the labeling at the time this appeared to be a huge problem. `detect_species` uses regular expressions to math species label along some other tricks. It returns CHERAB's `Element` object and species charge as a number (0 for neutrals).

```
species, charge = detect_species(
    structure=core_profiles_ids.ids.profiles_1d[0].ion[0],
    ion=True
)
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).

`ion` set to `True` for ion species or to `False` for neutrals. It changes regular expression pattern and sets `charge` to 0 for neutrals.

`distribution_density`

```
n_d = distribution_density(
    structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
    symbol="D",
    charge=1,
    psi_normalised=psi_normalised,
```

```
        equilibrium=equilibrium
    )
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).
`symbol` is a species symbol. It is used for messages.
`charge` is a species charge. It is used for messages.
`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.
`equilibrium` is an EFITEquilibrium object (section B.3.2).

This function checks if density profile stored in the IDS is correct: not empty, greater than zero, not equal to zero and not equal to 1.0. If it is not correct than message is produced and this species is not included in the plasma model.

`distribution_temperature`

```
    t_d = distribution_temperature(
        structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
        symbol="D",
        charge=1,
        psi_normalised=psi_normalised,
        equilibrium=equilibrium,
        t_average=core_profiles_ids.ids.profiles_1d[i].t_i_average,
        t_electrons=core_profiles_ids.ids.profiles_1d[i].electrons.temperature
    )
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).
`symbol` is a species symbol. It is used for messages.
`charge` is a species charge. It is used for messages.
`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.
`equilibrium` is an EFITEquilibrium object (section B.3.2).
`t_average` is an average ion temperature profile stored is the IDS. It is used in case if ion temperature profile for requested species is absent.
`t_electrons` is an electron temperature profile. It is used in case if ion temperature profile for requested species is absent.

`distribution_velocity`

```
    v_d = distribution_velocity(
        structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
        symbol="D",
```

```
        charge=1,
        psi_normalised=psi_normalised,
        equilibrium=equilibrium,
    )
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).
`symbol` is a species symbol. It is used for messages.
`charge` is a species charge. It is used for messages.
`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.
`equilibrium` is an EFITEquilibrium object (section B.3.2).

B.3.4 `edge_profiles` IDS

File `edge_profiles.py` contains `EdgeProfilesIDS` class which is used to read data from `edge_profiles` IDS and create CHERAB's `Plasma` object via `create_plasma` method. It is poses as edge plasma model.

Load a `edge_profiles` IDS:

```
edge_profiles_ids = EdgeProfilesIDS(
    shot=134000,
    run=30,
    user="public",
    database="iter",
    version="3",
)
```

`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,
`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,

Create a `Plasma` object:

```
edge_plasma = edge_profiles_ids.create_plasma(
    time=0.0,
    equilibrium=equilibrium,
    integration_step=0.001,
    integration_samples=5,
    parent=world,
    transform=None,
```

```
        name="Edge Plasma"
    )
```

`time` is a requested time;
`equilibrium` is an EFITEquilibrium object (section B.3.2);
`integration_step` is step of volumetric integration in meters;
`integration_samples` is a number of integration samples;
`parent` is a parent node;
`transform` is transformation matrix;
`name` is a name of this plasma.
Note: for more information on `integration_step` and `integration_samples` refer to CHERAB's documentation on `Plasma`. For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

`distribution_density`

```
    n_d = distribution_density(
        structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
        symbol="D",
        charge=1,
        psi_normalised=psi_normalised,
        equilibrium=equilibrium,
        interpolator=interp_options[interp],
        interpolation_data=interp_data[interp],
        mesh_lookup=te_mesh_lookup
    )
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).
`symbol` is a species symbol. It is used for messages.
`charge` is a species charge. It is used for messages.
`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.
`equilibrium` is an EFITEquilibrium object (section B.3.2).
`interpolator` – select an interpolator based on where values are defined: faces or nodes.
`interpolation_data` – select a function to process data based on where values are defined: faces or nodes.
`mesh_lookup` *empty*

All `interpolator`, `interpolation_data` and `mesh_lookup` are defined in the body of `create_plasma` method.

This function checks if density profile stored in the IDS is correct: not empty, greater than zero, not equal to zero and not equal to 1.0. If it is not correct than message is produced and

this species is not included in the plasma model.

`distribution_temperature`

```
t_d = distribution_temperature(
    structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
    symbol="D",
    charge=1,
    psi_normalised=psi_normalised,
    equilibrium=equilibrium,
    interpolator=interp_options[interp],
    interpolation_data=interp_data[interp],
    mesh_lookup=te_mesh_lookup
    t_average=core_profiles_ids.ids.profiles_1d[i].t_i_average,
    t_electrons=core_profiles_ids.ids.profiles_1d[i].electrons.temperature
)
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).

`symbol` is a species symbol. It is used for messages.

`charge` is a species charge. It is used for messages.

`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.

`equilibrium` is an EFITEquilibrium object (section B.3.2).

`interpolator` – select an interpolator based on where values are defined: faces or nodes.

`interpolation_data` – select a function to process data based on where values are defined: faces or nodes.

`mesh_lookup` *empty*

`t_average` is an average ion temperature profile stored is the IDS. It is used in case if ion temperature profile for requested species is absent.

`t_electrons` is an electron temperature profile. It is used in case if ion temperature profile for requested species is absent.

`distribution_velocity`

```
v_d = distribution_velocity(
    structure=core_profiles_ids.ids.profiles_1d[i].ion[j],
    symbol="D",
    charge=1,
    psi_normalised=psi_normalised,
    equilibrium=equilibrium,
    interpolator=interp_options[interp],
```

```
        interpolation_data=interp_data[interp],
        mesh_lookup=te_mesh_lookup
    )
```

`structure` is an IDS structure containing information on species (`ids.profiles_1d[i].ion[j]` or `ids.profiles_1d[i].neutral[j]`).
`symbol` is a species symbol. It is used for messages.
`charge` is a species charge. It is used for messages.
`psi_normalised` is a Ψ_{norm} profile. It is used to map density values.
`equilibrium` is an EFITEquilibrium object (section B.3.2).
`interpolator` – select an interpolator based on where values are defined: faces or nodes.
`interpolation_data` – select a function to process data based on where values are defined: faces or nodes.
`mesh_lookup` *empty*

`detect_species` is a copy of a function described in section B.3.3.

B.3.5 `charge_exchange` IDS

File `charge_exchange.py` contains `ChargeExchangeIDS` class which is used to read data from `charge_exchange` IDS and create different types of observers.

Load an `charge_exchange` IDS:

```
    charge_exchange_ids = ChargeExchangeIDS(
        shot=134000,
        run=30,
        user="public",
        database="iter",
        version="3",
    )
```

`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,
`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,

B.3.6 `nbi` IDS

File `nbi.py` contains `NBIIDS` class which is used to read data from `nbi` IDS and create CHERAB's `Beam` object via `create_beam` method. It is poses as DNB model.

Load an `nbi` IDS:

```
    nbi_ids = NBIIDS(
        shot=134000,
        run=30,
        user="public",
        database="iter",
        version="3",
    )
```

`shot` is an IMAS database shot ID,
`run` is an IMAS data run ID,
`user` is an IMAS database user ID,
`database` is an IMAS database ID,
`version` is an IMAS major version number,

```
    beam = nbi_ids.create_beam(
        time=0.0,
        plasma=core_plasma,
        atomic_data=adas,
        attenuation_instructions=attenuation_instructions,
        emission_instructions=emission_instructions,
        length=4.0,
        integration_step=0.001,
        integration_samples=5,
        parent=world
        transform=None,
        name="DNB"
    )
```

`time` is a requested time;
`plasma` is CHERAB's `Plasma` object;
`atomic_data` is modified CHERAB's `OpenADAS` object;
`attenuation_instructions` is a dictionary with attenuation instructions;
`emission_instructions` is a dictionary with emission instructions (see section B.7.3);
`integration_step` is step of volumetric integration in meters;
`integration_samples` is a number of integration samples;
`parent` is a parent node;
`transform` is transformation matrix;
`name` is a name of this plasma.
Note: for more information on `integration_step` and `integration_samples` refer to CHERAB's documentation on `Beam`. For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

At the time `Beam` supports model with only one beamlet and `create_beam` is designed with this in mind.

B.4 `machine` module – Setting the Wall

File `cxrs/machine/pfc_mesh.py` contains function `load_pfc_mesh` that is used to create a reactor wall model. All meshes are stored in `machine/portplugs` and `machine/simple` subfolders.

```
wall = load_pfc_mesh(
    path=os.path.join(os.path.dirname(__file__)),
    reflections=True,
    roughness={"Be": 0.26, "W": 0.29, "Ss": 0.13},
    detailed=False,
    parent=world,
    transform=None,
    name="Wall",
)
```

`path` is a path to .rsm mesh files;
`reflections` sets reflections on or off;
`roughness` is a roughness dictionary for PFC materials ("Be", "W", "Ss");
`detailed` – Load the detailed wall model instead of the simple one;
`parent` is a parent node;
`transform` is a transformation matrix;
`name` is a name of this plasma.
Here `reflections` assigns material properties to appropriate wall segments if set to `True` and sets the wall as perfect absorber if set to `False`. It is used to effectively turn reflections on and off. `roughness` argument sets *roughness* of the materials assigned to the wall segments (beryllium, tungsten and stainless steel). For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

B.5 `observers` module

Module `cxrs/observers` contains several files defining different observer classes built upon Raysect's observers.

B.5.1 Base Class

File `observers/base` contains `ObserverGroup` class that represents a group of observers, for example CXRS sightlines. The class has `observe` method that is used to perform an observation

by all observers in a group one by one and store the results. Methods `display` and `savetxt` are used to show registered spectra as an image or plot and save results in the text format respectively. They should be implemented in subclasses.

Example:

```
ObserverGroup(
    charge_exchange_ids=charge_exchange_ids,
    config="config.xml",
    wavelength_range=1,
    relative_error=0.05,
    scenario=scenario,
    parent=world,
    transform=None,
    name="CXRS Edge Sightlines"
)
```

`charge_exchange_ids` is a `ChargeExchangeIDS` object (section B.3.5),
`config` is a path to configuration file,
`wavelength_range` is a number representing a wavelength range for observation (it is defined in a configuration file),
`relative_error` is a minimal value of a desired relative error (if achieved relative error is higher than this value, number of pixel samples will be increased and observation will be performed again until desired relative error is achieved),
`scenario` is a dictionary containing all simulation labels: shot number, run number, time, used emission types, etc.,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

B.5.2 Sightlines

```
sightlines = SightlineGroup(
    ids=charge_exchange_ids,
    config="config.xml",
    wavelength_range=1,
    relative_error=0.05,
    scenario=scenario,
    parent=world,
    transform=None,
```

```
        name="CXRS Sightlines",
    )
```

`charge_exchange_ids` is a `ChargeExchangeIDS` object (section B.3.5),
`config` is a path to configuration file,
`wavelength_range` is a number representing a wavelength range for observation (it is defined in a configuration file),
`relative_error` is a minimal value of a desired relative error (if achieved relative error is higher than this value, number of pixel samples will be increased and observation will be performed again until desired relative error is achieved),
`scenario` is a dictionary containing all simulation labels: shot number, run number, time, used emission types, etc.,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

Methods:

`display` used to show and save produced image(s):

```
SightlineGroup.display(
    show=True,
    save=True,
    filename="cxrs_sightlines",
    dirname="output"
)
```

`show` – show produced images in separate window;
`save` – save images to a disk;
`filename` – name of the file to save to. Image is saved in .png format;
`dirname` – name of the directory to save to.

`savetxt` used to save simulation results in text format:

```
SightlineGroup.savetxt(
    filename="cxrs_sightlines",
    dirname="output"
)
```

`filename` – name of the file to save to. Text is saved in .txt format;
`dirname` – name of the directory to save to.

`draw_scheme` used to draw a simple scheme of a diagnostic:

```
SightlineGroup.draw_scheme(
    plane="xy",
    save=True,
```

```
    filename="cxrs_sightlines",
    dirname="output"
)
```

`plane` – produce a scheme in x-y ("xy") or r-z ("rz") plane;
`save` – save images to a disk;
`filename` – name of the file to save to. Image is saved in .png format;
`dirname` – name of the directory to save to.

B.5.3 Optics

```
optics = OpticsAssembly(
    target_distance=4.0,
    aperture_radius=0.05,
    magnification=0.1,
    refractive_index=1.52,
    parent=None,
    transform=None,
    name="Lens",
)
```

`target_distance` – distance to a target in meters;
`aperture_radius` – radius of the aperture in meters;
`magnification` – lens' magnification;
`refractive_index` – refractive index of the lens' material;
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

For more information on `parent`, `transform` and `name` refer to Raysect's documentation on `Node`.

B.5.4 Fibres

```
fibres = FibreGroup(
    ids=charge_exchange_ids,
    config="config.xml",
    wavelength_range=1,
    relative_error=0.05,
    scenario=scenario,
    parent=world,
    transform=None,
```

```
        name="CXRS Fibres",
    )
```

`charge_exchange_ids` is a `ChargeExchangeIDS` object (section B.3.5),
`config` is a path to configuration file,
`wavelength_range` is a number representing a wavelength range for observation (it is defined in a configuration file),
`relative_error` is a minimal value of a desired relative error (if achieved relative error is higher than this value, number of pixel samples will be increased and observation will be performed again until desired relative error is achieved),
`scenario` is a dictionary containing all simulation labels: shot number, run number, time, used emission types, etc.,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

Methods:

`display` used to show and save produced image(s):

```
FibreGroup.display(
    show=True,
    save=True,
    filename="cxrs_sightlines",
    dirname="output"
)
```

`show` – show produced images in separate window;
`save` – save images to a disk;
`filename` – name of the file to save to. Image is saved in .png format;
`dirname` – name of the directory to save to.

`savetxt` used to save simulation results in text format:

```
FibreGroup.savetxt(
    filename="cxrs_sightlines",
    dirname="output"
)
```

`filename` – name of the file to save to. Text is saved in .txt format;
`dirname` – name of the directory to save to.

`draw_scheme` used to draw a simple scheme of a diagnostic:

```
FibreGroup.draw_scheme(
    plane="xy",
    save=True,
```

```
        filename="cxrs_sightlines",
        dirname="output"
    )
```

`plane` – produce a scheme in x-y ("xy") or r-z ("rz") plane;
`save` – save images to a disk;
`filename` – name of the file to save to. Image is saved in .png format;
`dirname` – name of the directory to save to.

B.5.5 Camera

```
ccd_camera = CCDCamera(
    config="config.xml",
    parent=world,
    transform=None,
    name="CXRS Fibres",
)
```

`config` is a path to configuration file,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

B.5.6 Other Observers

```
fibre_bundle = FibreBundle(
    core_radius=0.5e-3,
    numerical_aperture=0.22,
    spectrum=,
    n_rows=3,
    n_cols=1,
    relative_error=0.05
    parent=world,
    transform=None,
    name="CXRS Fibres",
)
```

`core_radius` – radius of a fibre's core in meters;
`numerical_aperture` – fibre's numerical aperture;
`spectrum` – Raysect's `Spectrum` object;
`n_rows` – number of rows;
`n_cols` – number of columns;

`relative_error` – desired relative error;
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

```
scanner = Scanner(
    ids=charge_exchange_ids,
    config="config.xml",
    range_vertical=0.2,
    step_vertical=0.01,
    relative_error=0.05,
    scenario=scenario,
    parent=world,
    transform=None,
    name="Scanner",
)
```

`charge_exchange_ids` is a `ChargeExchangeIDS` object (section B.3.5),
`config` is a path to configuration file,
`range_vertical` – vertical range in meters;
`step_vertical` – size of the scanning step in vertical direction;
`relative_error` is a minimal value of a desired relative error (if achieved relative error is higher than this value, number of pixel samples will be increased and observation will be performed again until desired relative error is achieved),
`scenario` is a dictionary containing all simulation labels: shot number, run number, time, used emission types, etc.,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

```
total_radiance = TotalRadianceSightlines(
    ids=charge_exchange_ids,
    config="config.xml",
    relative_error=0.05,
    scenario=scenario,
    parent=world,
    transform=None,
    name="Scanner",
)
```

`charge_exchange_ids` is a `ChargeExchangeIDS` object (section B.3.5),

`config` is a path to configuration file,
`relative_error` is a minimal value of a desired relative error (if achieved relative error is higher than this value, number of pixel samples will be increased and observation will be performed again until desired relative error is achieved),
`scenario` is a dictionary containing all simulation labels: shot number, run number, time, used emission types, etc.,
`parent` is a parent node in a scenegraph,
`transform` is a transformation matrix,
`name` is a name for this group of observers.

```
spectrometer = Spectrometer(
    pixel_size=,
    n_pixels=,
    wl_calibration=,
    int_calibration=,
    width=,
    inst_func="rect",
    transmission=1.0,
)
```

`pixel_size` ?
`n_pixels` ?
`wl_calibration` ?
`int_calibration` ?
`width` ?
`inst_func` ?
`transmission` ?
Methods: ?

B.6 `populate` module

```
populate/create
```

```
populate_more(
    download=True,
    repository_path=None,
    adas_path="/work/projects/adas/adas/"
)
```

`download` – attempt to download the ADAS files if missing;
`repository_path` – alternate path for the OpenADAS repository;

`adas_path` – alternate path in which to search for ADAS files;

`populate/openadas`

```
adas = OpenADAS(
    data_path=None,
    permit_extrapolation=False,
    missing_rates_return_null=False
)
```

`data_path` – path to atomic data repository;

`permit_extrapolation` – if `True` informs interpolation objects to allow extrapolation beyond the limits of the tabulated data;

`missing_rates_return_null` – if `True`, allows `Null` rate objects to be returned when the requested atomic data is missing.

B.7 `utility` module

B.7.1 Getting Information on Plasma Parameters

This module contains routines used by `cxrs info` subprogram.

```
info/
├── __init__.py
├── beam.py
├── equilibrium.py
├── passive.py
├── plasma.py
└── profiles.py
```

`beam.py` file contains `beam_info` function designed to produce DNB related information for a model built for a simulation.

```
beam_info(
    plasma=core_plasma,
    id_dict = {
        "shot": 134000,
        "run": 30,
        "user": "public",
        "database": "iter",
        "version": "3",
    },
    plot_dict={
```

```
            "xlims": (4.0, 8.5),
            "ylims": (-4.5, 4.5),
            "resolution": 0.01,
            "beam_slice": 0.5,
        },
        dirname="beam_info",
        config="config.xml",
    )
```

`plasma` cherab's Plasma object;
`id_dict` dictionary with IMAS IDs;
`plot_dict` dictionary with plot parameters;
`dirname` path to save directory;
`config` path to configuration file.

`equilibrium.py` file contains `equilibrium_info` function designed to produce equilibrium related information for a model built for a simulation.

```
    equilibrium_info(
        id_dict = {
            "shot": 134000,
            "run": 30,
            "user": "public",
            "database": "iter",
            "version": "3",
        },
        plot_dict={
            "xlims": (4.0, 8.5),
            "ylims": (-4.5, 4.5),
            "resolution": 0.01,
            "beam_slice": 0.5,
        },
        dirname="equilibrium_info",
        filename=None,
    )
```

`id_dict` dictionary with IMAS IDs;
`plot_dict` dictionary with plot parameters;
`dirname` path to save directory;
`filename` name of the produced file.

`passive.py` file contains `passive_info` function designed to produce passive charge-exchange related information for a model built for a simulation.

```
passive_info(
    plasma=core_plasma,
    beam=beam,
    id_dict = {
        "shot": 134000,
        "run": 30,
        "user": "public",
        "database": "iter",
        "version": "3",
    },
    plot_dict={
        "xlims": (4.0, 8.5),
        "ylims": (-4.5, 4.5),
        "resolution": 0.01,
        "beam_slice": 0.5,
    },
    dirname="passive_info",
)
```

`plasma` CHERAB's `Plasma` object;
`beam` CHERAB's `Beam` object;
`id_dict` dictionary with IMAS IDs;
`plot_dict` dictionary with plot parameters;
`dirname` path to save directory;

`profiles.py` file contains `profiles_info` function designed to produce plasma profiles.

```
profiles_info(
    core_plasma=core_plasma,
    edge_plasma=edge_plasma,
    beam=beam,
    id_dict = {
        "shot": 134000,
        "run": 30,
        "user": "public",
        "database": "iter",
        "version": "3",
```

```
    },
    plot_dict={
        "xlims": (4.0, 8.5),
        "ylims": (-4.5, 4.5),
        "resolution": 0.01,
        "beam_slice": 0.5,
    },
    dirname="profiles",
)
```

`core_plasma` CHERAB's `Plasma` object for core plasma;
`edge_plasma` CHERAB's `Plasma` object for edge plasma;
`beam` CHERAB's `Beam` object;
`id_dict` dictionary with IMAS IDs;
`plot_dict` dictionary with plot parameters;
`dirname` path to save directory;

B.7.2 Parsing XML Configuration File

File `utility/xml.py` contains functions to work with configuration files. For XML parsing module `xml.etree` from Python's standard library is used.

`srt2bool(string)`

converts "on" to `True` and "off" to `False`.

`bool2str(arg)`

converts `True` to "on" and `False` to "off".

`read_xml_entry(tree_element)`

reads XML structure and returns Python object based on structure's description.

`parse_user_options(section, config)`

reads `<user_options>` `section` from the configuration file and returns dictionary with its parameters.

`parse_emission_lines(config)`

reads `<emission_lines>` section of the configuration file and returns list of CHERAB's `Line` objects.

`parse_diagnostic_geometry(config)`

reads `<geometry> section` from the configuration file and returns dictionary with its parameters.

`parse_dnb_parameters(config)`

reads `<DNB> section` from the configuration file and returns dictionary with its parameters.

`parse_wavelength_ranges(config)`

reads `<wavelength_ranges> section` from the configuration file and returns dictionary with its parameters.

B.7.3 Setting Emission Parameters

```
plasma_emission_parameters = plasma_emission(
    plasma,
    lines,
    bremsstrahlung=True,
    recombination=True,
    excitation=True,
    passive=True,
)
```

`plasma` cherab's Plasma object;
`lines` list of cherab's Line objects;
`bremsstrahlung` turn Bremsstrahlung on;
`recombination` turn recombination emission on;
`excitation` turn excitation emission on;
`passive` turn passive emission on.

```
beam_emission_parameters = beam_emission(
    plasma,
    lines,
)
```

`plasma` cherab's Plasma object;
`lines` list of cherab's Line objects;

B.7.4 Math Functions

`to_cylindrical(point)`

convert Raysect's Point3D from cartesian coordinates to cylindrical.

`to_cylindrical_multiple(points)`

convert list of Raysect's Point3D from cartesian coordinates to cylindrical.

`to_cartesian(point)`

convert Raysect's Point3D from cylindrical coordinates to cartesian.

`to_cartesian_multiple(points)`

convert list of Raysect's Point3D from cylindrical coordinates to cartesian.

```
ion_temperature(
    doppler_width,
    natural_wavelength,
    atomic_weight
)
```

Calculate impurity's ion temperature by Doppler width of observed spectral line: $T_\mathrm{i} = mc^2 \left(\frac{\Delta\lambda}{\lambda_0}\right)^2$
`doppler_width` width of line's Doppler broadening in nm;
`natural_wavelength` line's "natural" wavelength in nm;
`atomic_weight` impurity's atomic weight.

```
velocity_edge(
    natural_wavelength,
    doppler_shift_upper,
    doppler_shift_lower,
    angle_upper,
    angle_lower,
)
```

Calculate plasma bulk velocity at the edge:

$$v_\mathrm{tor} = \frac{c}{\lambda_0} \frac{\Delta\lambda_\mathrm{upper} \sin(\alpha_\mathrm{lower}) + \Delta\lambda_\mathrm{lower} \sin(\alpha_\mathrm{upper})}{\sin(\alpha_\mathrm{upper} + \alpha_\mathrm{lower})}$$
$$v_\mathrm{pol} = \frac{c}{\lambda_0} \frac{\Delta\lambda_\mathrm{upper} \cos(\alpha_\mathrm{lower}) - \Delta\lambda_\mathrm{lower} \cos(\alpha_\mathrm{upper})}{\sin(\alpha_\mathrm{upper} + \alpha_\mathrm{lower})}$$

`natural_wavelength` line's "natural" wavelength in nm;
`doppler_shift_upper` Doppler shift in nm registered by upper diagnostic;
`doppler_shift_lower` Doppler shift in nm registered by lower diagnostic;
`angle_upper` angle in rad between upper diagnostic's line of sight and toroidal direction;
`angle_lower` angle in rad between lower diagnostic's line of sight and toroidal direction;

```
velocity_core(
    doppler_shift
    angle,
    natural_wavelength,
)
```

Calculate plasma bulk velocity at the edge: $v = \frac{c}{\cos(\alpha)} \frac{\Delta\lambda}{\lambda_0}$
`doppler_shift` Doppler shift in nm registered by diagnostic;
`angle` angle in rad between diagnostic's line of sight and toroidal direction;
`natural_wavelength` line's "natural" wavelength in nm.

```
lines_intersection(
    start_a,
    vector_a,
    start_b,
    vector_b,
    cylindrical=False,
)
```

Find point of intersection between two lines. Lines are defined by starting point and direction vector: $\vec{r} = \vec{r_0} + \vec{a}t$.
`start_a` start point of the first line;
`vector_a` direction vector of the first line;
`start_b` start point of the second line;
`vector_b` direction vector of the second line;
`cylindrical` return intersection point in cylindrical coordinates.

```
line_cylinder_intersection(
    start,
    vector,
    radius,
    cylindrical=False
)
```

Find point of intersection between a line and a cylinder. Line is defined by starting point and direction vector: $\vec{r} = \vec{r_0} + \vec{a}t$. Cylinder's axis coincides with Z axis.

`start` start point of the line;
`vector` direction vector of the line;
`radius` radius of the cylinder;
`cylindrical` return intersection point in cylindrical coordinates.

```
tangency_to_cartesian(
    point,
    tangency_radius,
    angle,
    direction
)
```

Convert NBI IDS beam direction parameters to vector in cartesian coordinates.
`point` Beam source point;
`tangency_radius` Tangency radius (major radius where the central line of a NBI unit is tangent to a circle around the torus) [m];
`angle` Angle of inclination between a beamlet at the centre of the injection unit surface and the horizontal plane [rad];
`direction` Direction of the beam seen from above the torus: -1 = clockwise; 1 = counter clockwise.

```
cartesian_to_tangency(point, vector)
```

Convert beam direction vector in cartesian coordinates to NBI IDS parameters.
`point` Beam source coordinates point [m];
`vector` Beam direction vector [m].

```
convolve(data, kernel)
```

Convolve without edge effects.
`data` Data array;
`kernel` Kernel array.

B.7.5 Fitting Routine

```
fit(
    lines,
    observer_groups,
    atomic_data,
    background=True,
    fit_report=False,
    dirname=None,
```

```
        scenario=None,
        save=True,
    )
```

Fit emission spectra and reconstruct plasma parameters.

`lines` List of beam emission lines.

`observer_groups` List of observer groups.

`atomic_data` Atomic data object.

`fit_report` If True, plot fit report. (Default value = False)

`dirname` Path to save directory. (Default value = None)

`scenario` Dictionary with scenario parameters. (Default value = None)

`save` If True, save results in image format. (Default value = True)

`find_nearest(array, value)`

Find index of the `array` element with value closest to `value`.

`background_model(wavelengths, spectrum)`

Define model for background (Bremsstrahlung) fitting: $B(\lambda) = A/\lambda$.

```
    reference_line_model(
        line_label,
        line_data,
        wavelengths,
        spectrum,
        bg_model=None
    )

    def line_model(
        line_label,
        line_data,
        reference_line_label,
        reference_line_data,
        wavelengths,
        spectrum,
        bg_model=None,
    )
```

B.7.6 Others

utility/annotation utility/data utility/timer utility/fit