ThunderBoltz

Release 0.1

Ryan Park, Brett Scheiner, Mark Zammit

Los Alamos National Laboratory, Los Alamos, NM, 87545

Oct 15, 2023

CONTENTS

This documentation includes some simple tutorials for using the ThunderBoltz plasma simulation package and a complete public API reference.

CHAPTER

ONE

INSTALLATION

For now, the code must be downloaded from a repository. Use the following command to clone the code into a local repository.

```
git clone git@gitlab.com/Mczammit/thunderboltz.git
```

You may need to set up SSH keys in order to access gitlab. See the Gitlab SSH Guide to set up access to GitLab repositories.

The basic ThunderBoltz functionality is available either as an executable in `bin/thunderboltz.bin` or can be compiled from the source in `src/thunderboltz`. To install the Python interface, run the `install.sh` script from the root directory to install the python package.

```
./install.sh
```

Warning: This will upgrade pip and install specific versions of python packages, so create an environment if you are concerned with python package overwrite.

CHAPTER

TWO

TUTORIALS

See the *Quick Start Guide* to go through a brief tutorial setting up a simple calculation and interpreting the results.

See *Preparing Cross Sections* for a tutorial on the various ways to obtain and manipulate cross sections for ThunderBoltz simulations.

See *Running Multiple Calculations* for a quick guide on how to vary simulation parameters and easily run simulations in parallel.

See *Extracting Results* for details on easily parsing, plotting, and exporting data from many ThunderBoltz simulations at once.

2.1 Quick Start Guide

2.1.1 Running the Simulation

In this guide we will set up a simple calculation, run the program, and interpret the results. This guide assumes that the python ThunderBoltz package has already been *Installed*.

For the first example, we will set up a Helium gas calculation, since the package already has a Helium model built-in. First, set up a file that will hold the python script to drive the simulation. In a new file `run_example.py`, write the following:

```
import os # Operating system interface for making directories

from pytb import ThunderBoltz # Import the main simulation object
from pytb.input import He_TB  # This is a built-in He model preset

# First we will make a folder in the current directory to house
# The simulation output / logging files.
os.makedirs("example_sim")

tb = ThunderBoltz(

    # Specify internal ThunderBoltz settings.
    DT=1e-10,             # Time step of 0.1 ns.
    NS=30000,             # Number of time steps.
    L=1e-6,               # The cell size (m).
    NP=[10000, 1000],     # 1e4 electrons, 1e3 Helium macroparticles.
    FV=[20000, 10000, 0], # Dump the electron velocities on steps 20000 and 30000.
    # ... etc.
```

(continues on next page)

(continued from previous page)

```
    # Specify additional python interface settings in the same way.
    Ered=100,       # Fix the reduced field at 100 Townshend.
    eesd="uniform", # Use the uniform electron energy sharing ionization model.
    eadf="default", # Use isotropic elastic scattering.
    egen=False,     # Do not generate secondary electrons in ionization events.
    # ... etc.


    ### The package comes with a built-in Helium model that can be controlled
    ### with the following parameters.

    # This indicates use of the built in He model. It will automatically
    # set up the masses, charges, and cross sections for a fixed-background
    # Helium calculation.
    indeck=He_TB,
    # You may also specify up to what principle quantum number the
    # excitation cross sections will go.
    n=4,
    # Or how many points to sample the cross section model.
    nsamples=300,

    # Finally, specify a simulation directory where logging and output
    # files will be written.
    directory="example_sim",
)


# At this point, the ThunderBoltz object has been configured and is ready
# to run. To do this, just call the "run" method. This will write the
# necessary input files, compile the program from package source into the
# simulation directory, and execute the program in a subprocess.
tb.run(
    # By default, all output is written into files and not stdout,
    # but data can also be printed to stdout by setting
    std_banner=True,
)

# Once the calculation is finished, your python code will continue
# executing, and you can extract data from the ThunderBoltz object.

# This will extract step by step data for reaction counts/rates,
# electron mobility, drift velocity, and more.
ts = tb.get_timeseries()
# This will extract the same quantities but time averaged during the
# steady state period of the run.
steady_state = tb.get_ss_params()

# Sometimes it is easier to extract data once it is already been
# calculated. The next section will demonstrate how to asynchronously
# extract data after the calculation has finished in a new process.
```

This script can be run from the command line by simply executing

```
python run_example.py
```

Warning: When running and rerunning calculations, ensure that the specified simulation directory has no output files already present. ThunderBoltz will not overwrite these files when running more calculations. This will preserve your data, but prevent the python interface from being able to interpret the results.

For a full list available ThunderBoltz parameters, see *Simulation Parameters*.

2.1.2 Interpreting the Results

Some calculations of interest may take several hours, and so it is beneficial to run it once and explore the data later. It is easy to recover the output data from the output files after the calculation is finished like so:

```
# Import some python plotting tools
import matplotlib.pyplot as plt

# This will read single calculations and return ThunderBoltz objects.
from pytb import read

# Ensure you are running this code from the same place as above, and
# just pass the location of the simulation directory.
tb = read("example_sim")

# Now all the same data will be available in the form of pandas DataFrames.
timeseries = tb.get_timeseries() # Returns timeseries data in a DataFrame.
steady_state = tb.get_ss_params() # Returns steady state data in a DataFrame.
velocity_data = tb.get_vdfs("all") # Returns all velocity dump data in a DataFrame.

# These frames are convenient because they can be easily manipulated and
# exported

# To export to csv:
timeseries.to_csv("example_sim/timeseries.csv", index=False)
steady_state.to_csv("example_sim/steady.csv", index=False)

# One can truncate the data row-wise. For example,
# the following will take data from last 20000 steps
# of the 30000 steps.
trunc = timeseries[timeseries.step >= 10000].copy()

# Or only look at certain columns

# This will extract the mean electron energy (MEe),
# the reduced electron mobility (mobN),
# and the Townshend ionization coefficient (a_n).
transport_params = timeseries[["MEe", "mobN", "a_n"]].copy()


# There are also some built-in plotting methods that
# can be accessed through the ThunderBoltz object.
```

(continues on next page)

(continued from previous page)

```
# This will plot step by step data for any of the output
# parameters available in the time series table. Default
# is mean energy, mobility, and Townshend ionization coefficient
tb.plot_timeseries()

# This will plot the rate coefficients for every process.
tb.plot_rates()

# This will plot a joint plot of the electron velocity distribution function
tb.plot_vdfs()

# This will show a GUI and is required to actually display the plots.
plt.show()
```

For more details on the output parameter format, see output_params.

2.2 Preparing Cross Sections

There are a variety of ways to specify cross sections with the ThunderBoltz interface. In the *Quick Start Guide*, we used a built-in Helium cross section model. A more general approach to preparing cross sections is with the *CrossSections* object.

2.2.1 Initializing the CrossSections Object

There are three main ways to initialize a *CrossSections* object:

1. With cross section data from another ThunderBoltz run

   ```
   from pytb import CrossSections
   # Just specify the path to the simulation directory of a
   # different ThunderBoltz run.
   cross_sections = CrossSections(input_path="path/to/thunderboltz_sim_dir")
   ```

   Refer to the *CrossSections* section of the *API Reference* to ensure the simulation data is set up correctly for interpretation by *CrossSections*.

1. By reading from an LXCat text file extract.

   ```
   from pytb import CrossSections
   # First initialize an empty cross sections object
   cross_sections = CrossSections()
   # Then reference a text file extract from LXCat
   cross_sections.from_LXCat("path/to/LXCat_data.txt")
   ```

   Note: For now, the LXCat parser assumes two species electron-gas systems where all processes are between electrons and gas macroparticles. If you wish to use LXCat data for other purposes, you can alter the species indices to your liking via `CrossSections.table` after loading in LXCat data.

2. By programmatically generating cross section data in python.

   This approach involves the *Process* object.

```
from pytb import CrossSections
from pytb import Process

# Initialize an empty cross sections object
cross_sections = CrossSection()

# Next make a few processes

# You can pass arbitrary tabulated data like so
elastic_data = [
  # [eV], [m^2]
    [0.0, 2e-20],
    [0.001, 2.1e-20],
    [.01, 3e-20],
    [10.0, 1e-19],
    [1000, 1e-18],
    [10000, 2e-19],
]
elastic_process = Process(
    "Elastic", # The type of process
    r1=0, # The first reactant species index
    r2=1, # The second reactant species index
    p1=0, # The first product species index
    p2=1, # The second product species index
    cs_data=elastic_data,
    # This will determine the name of the
    # written cross section file and ideally should
    # be unique.
    name="elastic_example",
)
# You can also pass data frames, or ndarrays if that is
# preferable

# Or, use an analytic form defined with a python
# function.
import numpy as np # Import math functionality
def inelastic_model(energy, parameter):
    # It's okay to have conditional statements
    if energy < 5:
        return parameter

    # And nonlinear functions
    return parameter*np.log(energy)/energy

# You can parameterize your model
cs_mod_1 = lambda e: inelastic_model(e, 1e-20)
cs_mod_2 = lambda e: inelastic_model(e, 2e-20)
cs_mod_3 = lambda e: inelastic_model(e, 3e-20)

# And create multiple cross sections
```

(continues on next page)

(continued from previous page)

```
inelastic_1 = Process(
    "Inelastic", threshold=1., cs_func=cs_mod_1, name="inelastic1")
inelastic_2 = Process(
    "Inelastic", threshold=1., cs_func=cs_mod_2, name="inelastic2")
inelastic_3 = Process(
    "Inelastic", threshold=1., cs_func=cs_mod_3, name="inelastic3")

# Finally, you can create processes with differential cross section
# models, if they are available in your ThunderBoltz version.
ionization = Process("Ionization", threshold=10.,
    cs_func=lambda e: 1e-19*np.log(e)/e,
    # This, for example, will add the equal energy sharing condition
    differential_process="equal",
    name="ionization")


# You can add your process to the CrossSections object one at a time
cross_sections.add_process(elastic_process)
# Or all at once
cross_sections.add_processes(
    [inelastic_1, inelastic_2, inelastic_3, ionization]
)
```

Note: It is important to explicitly specify threshold values for inelastic and superelastic processes because their values will not be inferred from the cross section data.

2.2.2 Viewing Your Cross Sections

When parsing data from external sources, it is important to ensure that the correct data is being used in the intended context for the simulation. You can view the reaction table for the model by printing out the `table` attribute.

```
print(cross_section.table)
```

And you can view the cross section data associated with each process by printing out the `data` attribute.

```
print(cross_section.data)
```

To view a plot of the cross section data, use the *plot_cs()* method.

```
cross_section.plot_cs()

# Remember to show the plot at the end of plotting scripts
# Make sure to include the import statement "import matplotlib.pyplot as plt"
plt.show()
```

See the API reference for plotting related quantities with the *plot_cs()* method.

2.2.3 Attaching the `CrossSections` Object

Finally, attach the `CrossSections` object to the main ThunderBoltz object using the `cs` keyword to use the cross section model within it.

```
tb = ThunderBoltz(
    # ...
    cs=cross_sections,
    # ...
)

tb.run()
# ...
```

2.3 Running Multiple Calculations

2.3.1 In Sequence

You can change simulation parameters in the *ThunderBoltz* object and run the program again in a new directory. Use the *set_()* method to update the desired parameters.

Suppose you wanted to run several calculation at various field values. To do this, loop through the field values, create new directories for the new calculation and run the object like so:

```
import os
import pytb

# Make a base directory for this ensemble of simulations
os.makedirs("multi_sim")

tb = ThunderBoltz(indeck=pytb.input.He_TB)

fields = [10, 100, 500]
# Loop through the field values
for field in fields:
    # Create a new directory for this calculation
    subdir = os.path.join("multi_sim", f"{field}Td")
    os.makedirs(subdir)
    tb.set_(Ered=field, directory=subdir)
    # Run the calculation
    tb.run()
```

Each call to *run()* will block until the corresponding simulation is finished.

2.3.2 In Parallel

Now suppose you would like to take advantage of multiple cores to run several ThunderBoltz calculations at once. Though the internal kinetic code is not (yet) parallelized, the python interface can run several ThunderBoltz subprocesses in parallel like so:

```
import os
import pytb

# Make a base directory for this ensemble of simulations
base_path = "multi_sim_parallel"
os.makedirs(base_path)

# Create the base object for the calculation
tb = ThunderBoltz(indeck=pytb.input.He_TB)

fields = [10, 100, 500]

# This time use the DistributedPool context,
# passing the ThunderBoltz object like so
with DistributedPool(tb) as pool:
    # Loop through the field values
    for field in fields:
        # Create a new directory for this calculation
        subdir = os.path.join(base_path, f"{field}Td")
        os.makedirs(subdir)

        # Rather than running with the ``ThunderBoltz`` object,
        # submit the changes to the pool, and it will automatically
        # run each each submitted calculation in parallel.
        pool.submit(Ered=field, directory=subdir)

# The DistributedPool context will wait for all the jobs to finish
# before continuing execution outside the 'with' block.
```

Warning: The forking process used to run multiple simulations has thusfar only been tested on UNIX/LINUX operating systems.

Warning: Ensure there is enough simultaneous memory for all jobs when running them in parallel. See the section on Electron Growth and Memory Management.

2.3.3 With a Job Manager

If HPC resources are available to the user, the python API includes a job manager compatible with the SLURM protocol. The *SlurmManager* context allows for many different calculations to be split up among compute nodes, and further distributed across cores. Use it as follows:

```
import os
import pytb

# Make a base directory for this ensemble of simulations
base_path = "multi_sim_slurm"
os.makedirs(base_path)

# Create the base object for the calculation
tb = ThunderBoltz(indeck=pytb.input.He_TB)

fields = [10, 100, 500]

# Configure SLURM parameters for your job
slurm_options = {
    "account": "my_account",
    "time": 100, # in minutes
    "job-name": "test_slurm",
    "ntasks-per-node": 8, # Specify number of cores to use
    "qos": "debug",
    "reservation": "debug",
}

# Use the SlurmManager Context, just like the DistributedPool context,
# but also give it your SLURM options.
with SlurmManager(tb, base_path, **slurm_options) as slurm:
    # Loop through the field values
    for field in fields:
        # Create a new directory for this calculation
        subdir = os.path.join(base_path, f"{field}Td")
        os.makedirs(subdir)
        # Use the slurm manager the same way as the pool, it will
        # handle node and core allocation internally.
        slurm.submit(Ered=field, directory=subdir)
```

See here for an explanation of the ** (unpacking) operator used in the previous example.

Note: This job manager currently only works for clusters that either already have the gcc and python requirements installed on each compute node, or clusters that use the Module System to load functionality.

The default behavior is to accomodate the module system as it is common on most HPC machines. If you wish to avoid writing `module load` commands in the SLURM script, simply specify `modules=[]` in the `SlurmManager` constructor.

Warning: Ensure there is enough memory for all parallel jobs when running them in parallel.

2.4 Extracting Results

2.4.1 Reading a Single Calculation

After a calculation is finished, you can easily read the output data using the python API like so:

```
import pytb

# Pass the location of the simulation directory to be read
tb = pytb.read("path/to/previous/simulation_directory")
```

A single *ThunderBoltz* object will be returned, with which you can easily *export* or *plot* output data.

When reading calculations in this way, you may or may not want to extract cross section data from the simulation directory as well. To save on runtime, cross section data is not read in by default. However, if you wanted to read in cross section data from an old calculation and reuse that data for other purposes, you can use the `read_cs_data` argument:

```
import pytb

tb = pytb.read("path/to/previous/simulation_directory", read_cs_data=True)

# You will now see the cross section data has been loaded
print(tb.cs.data)
```

2.4.2 Reading Many Calculations

When running many calculations in various directories, it can be convenient to read all of the output data at once. Imagine a directory structure like this:

```
path/to/base_path
 /---sim1
     /---indeck_file.in
     /---cross_sections
     /---thunderboltz.out
     ...
 /---sim2
     ...
 /---sim3
     ...
 ...
```

where several ThunderBoltz calculations are stored in one base directory located at `path/to/base_path`. You locate and extract all relevant ThunderBoltz data out of a directory tree using the *query_tree()* function:

```
import pytb

tbs = pytb.query_tree("path/to/base_path")

# Now you can access each of the simulation objects
# separately. For example:
```

(continues on next page)

(continued from previous page)

```
# View the time series data from the first read calculation.
print(tbs[0].get_timeseries)

# Plot the last velocity dump data from the fourth read calculation.
tbs[3].plot_vdfs()
```

See the *query_tree()* API reference to learn about options for filtering criteria and automatically merging data from several calculations.

As with the *single calculation* case, you can request the cross section data by providing the `read_cs_data` argument:

```
import pytb

tbs = pytb.query_tree("path/to/previous/simulation_directory", read_cs_data=True)

# Now each of the calculations will have cross section model data attached to them.
# For example, this will print the collision table for the 3rd read in calculation.
print(tb[2].cs.table)
```

2.4.3 Accessing Data

Either after a calculation has finished, or after reading output data as shown above, all data can be extracted from the *ThunderBoltz* object:

Time-dependent data for the attributes found in *OutputParameters* can be accessed with *get_timeseries()*:

```
data = tb.get_timeseries()
```

Time-averaged data for the attributes found in *OutputParameters* can be accessed with *get_ss_params()*:

```
data = tb.get_ss_params()
```

This method will also compute standard deviations over the steady-state interval for each parameter in a new column with a "_std" suffix added to the column name.

Warning: Currently, the last quarter of the timeseries data is assumed to be in steady-state by default when calculating these steady-state parameters. Please verify that this is true by viewing the figures produced by `plot_timeseries()`. Otherwise, run the simulation for longer, or provide your own appropriate criteria via the `ss_func` option when calling *get_ss_params()*.

Output parameters for the attributes found in *ParticleParameters* can be accessed with *get_particle_tables()*:

```
data = tb.get_particle_tables()

# For example, this will write the mean energy, and
# each of the mean displacement components to a csv
# called "R_export.csv"
data.to_csv("R_export.csv", index=False)
```

2.4.4 Exporting Data

Once data is in the form of a `DataFrame`, it is easy to export it to other formats. See the Pandas I/O Guide for extensive options for converting from the `DataFrame` object. The simplest option is to convert the data to a csv:

```
# This will write the data into a new file called "my_new_file.csv"
data.to_csv("my_new_file.csv", index=False)
```

Note: When exporting data to the csv format from a pandas DataFrame, it is usually most convenient to pass `index = False` to prevent `to_csv()` from writing the index (usually just an enumeration of the rows) into the first column of the csv.

2.4.5 Plotting Results

The *ThunderBoltz* API offers functions for automatically plotting results. See the documentation for the following functions

pytb.ThunderBoltz.plot_timeseries([series, ...])	Create a diagnostic plot of ThunderBoltz time series data.
pytb.ThunderBoltz.plot_rates([save, stamp, ...])	Create a diagnostic plot of ThunderBoltz time series data.
pytb.ThunderBoltz.plot_edf_comps([steps, ...])	Plot the directional components of the energy distribution function.
pytb.ThunderBoltz.plot_edfs([steps, ...])	Plot the electron total energy distribution function, optionally include the provided cross sections for comparison.
pytb.ThunderBoltz.plot_cs([ax, legend, ...])	Plot the cross sections models.

These functions will plot the data into `Figure` objects, but in order to see the plots in a GUI, you must import the plotting library and include the line `plt.show()` after calling plotting methods like so:

```
import pytb

# This will import the plotting library
import matplotlib.pyplot as plt

# Either read in data, or run calculations
tb = pytb.read("path/to/simulations_to_plot", read_cs_data=True)

# Call plotting methods
tb.plot_cs()

# Show the plots and load a GUI
plt.show()
```

Alternatively, you may specify a directory within which to save a pdf file of the plot when calling any `ThunderBoltz.plot_*` method.

```
tb.plot_cs(save="path/to/figure_directory")
```

CHAPTER

THREE

BENCHMARK TESTING

See *Benchmark Testing* to run code that reproduces the results found in the paper.

3.1 Benchmark Testing

There are three benchmark tests available in the repository. These benchmark simulations are described in detail in Sect. III of the ThunderBoltz paper. The resulting calculations and figures from these benchmark tests can be compared directly to the figures given in the paper. Each can be imported from the `run.py` python module.

3.1.1 Onsager Relation

The Onsager relation[1] predicts the kinetic rates of the following chemical reactions between arbitrary heavy particles,

$$A \rightleftarrows B \rightleftarrows C \rightleftarrows A.$$

Based on the equilibrium condition, $n_i k_{ij} = n_j k_{ji}$, the rate constants k_{ij} have the analytic solution

$$k(T) = 2d^2 \left(\frac{2\pi k_B T}{m_r}\right)^{1/2} \exp\left(\frac{-E_a}{k_B T}\right)\left(1 + \frac{E_a}{k_B T}\right).$$

To run this system in in ThunderBoltz, run the prepared function either in a python script:

```
# Run this from within the repository root
import run
run.onsager_relation()
```

or directly from the command line:

```
python -c "import run; run.onsager_relation()"
```

The resulting calculation will automatically run in the directory `simulations/onsager_relation`.

Once the simulation has finished, run the following on the command line (or in a python script) to view a time evolution of the species densities, reaction rates, and absolute rates.

```
python -c "import visualize; visualize.plot_onsager()"
```

This will automatically save a pdf of the plot in the `simulations` directory.

[1] Light, J. C., Ross, J., & Shuler, K. E. (1969). Rate coefficients, reaction cross sections and microscopic reversibility. Kinetic Processes in Gases and Plasmas, 314, 281.

3.1.2 Ikuta-Sugai

The Ikuta-Sugai benchmark problem tests electron transport in crossed electric and magnetic fields.

To run this system in ThunderBoltz and compare it to the analytic theory presented by Ness[2], run the prepared function either in a python script of directly from the command line:

```
python -c "import run; run.ikuta_sugai()"
```

Once the simulation has finished, run the following command to view the effect of the magnetic field on the average velocity moments and mean energy of the particles in comparison to Ness:

```
python -c "import visualize; visualize.plot_ikuta_sugai()"
```

This will automatically save a pdf of the plot in the `simulations` directory.

3.1.3 He Transport

Here we generate comparisons of bulk and flux electron mobility, μN, and Townshend ionization coefficient, α/N, at various reduced fields. We compare ThunderBoltz results to the two-term Boltzmann equation solver, BOLSIG, as well as some swarm experiments.

To simulate this system in ThunderBoltz run the prepared function either in a python script or directly from the command line:

```
python -c "import run; run.He_transport()"
```

Once the simulation has finished, run the following command to view the reduced Townshend ionization coefficient and the reduced electron mobility as a function of reduced electric field:

```
python -c "import visualize; visualize.plot_He_transport()"
```

This will automatically save a pdf of the plot in the `simulations` directory. To view a plot comparing the individual reaction rate coefficients of ThunderBoltz and BOLSIG, run the following:

```
python -c "import visualize; visualize.rate_comp()"
```

[2] K F Ness 1994 J. Phys. D: Appl. Phys. 27 1848.

CHAPTER

FOUR

SIMULATION PARAMETERS

Review the *Simulation Parameters* for information on the default behavior of the code, the available input options, and details regarding output parameter definitions and interpretations.

4.1 Simulation Parameters

4.1.1 Input Parameters

pytb.parameters.TBParameters()	The ThunderBoltz simulation settings and their default values.
pytb.parameters.WrapParameters()	Additional Python interface settings and their defaults.

pytb.parameters.TBParameters

class pytb.parameters.**TBParameters**

The ThunderBoltz simulation settings and their default values.

Attributes

B	(list[int]) Magnetic field vector (Tesla), default is `[0.0, 0.0, 0.0]`.
CO	(str) Collision ordering, options are `"default"` | `"Random"` | `"Reverse"`.
CR	(int) If 1, then the remainder of N_{pairs} is carried into the next N_{pairs} evaluation of the same process, default is 1.
DT	(float) Time increment interval (s), default is `5e-12`.
E	(float) Electric field in z-direction (V/m), default is `-24640.0`.
ET	(float) E-field oscillation frequency (Hz), default is `0`.
EX	(int) Cross section extrapolation — options are `0` (extrapolated cross sections are set to 0 m ^{2}), or 1 (linearly extrapolated from last two points), default is `0`.
FV	(list[int]) Output velocity dump settings [start, stride, species ID], default is `[1000, 1000000, 0]`.
L	(float) Cell length (m), default is `1e-6`.
LV	(list[str,int]) Optionally load particle velocities from a comma separated text file, default is `None`; specify the name of the file at index `0` and the particle species index it applies to at index 1.
MEM	(float) Request memory (GB) for particle arrays.
MP	(list[float]) Mass of each particle species (amu), default is `[5.4857e-4, 28.0]`.
NP	(list[int]) Number of particles for each species, default is `[10000, 1000]`.
NS	(int) Number of time steps, default is `1000001`.
OS	(int) Time step stride for output parameters, default is `100`.
QP	(list[int]) Charge (elementary units) of each particle species, default is `[-1.0, 0.0]`.
SE	(int) When using a SLURM manager on HPC, auto dump particle velocity data before job allocation runs out — options are `0` (don't auto dump) | 1 (dump using SLURM setup).
SP	(int) Number of species, default is `2`.
TP	(list[float]) Temperature (eV) of each particle species, default is `[0.0, 0.0259]`.
VS	(int) Number of random samples used to find $\max_\epsilon(v\sigma(\epsilon))$ for each process, default is `1000`.
VV	(list[float]) Flow velocity for each particle, default is `[0.0, 0.0]`.

Methods

get_params()	Return the set of parameters and their default values as a python dictionary.

pytb.parameters.TBParameters.get_params

TBParameters.**get_params**()

Return the set of parameters and their default values as a python dictionary.

pytb.parameters.WrapParameters

class pytb.parameters.**WrapParameters**

Additional Python interface settings and their defaults. An asterisk (*) at the beginning of the description indicates a parameter that is specific to the built-in He model.

Attributes

Bred	(list[float]) Specify reduced magnetic field (Hx), default is None.
DE	(float) The maximum change in energy (eV) of a hypothetical electron per time step, default is 0.1.
ECS	*(str or None) The total elastic cross section model for the He built-in — options are "ICS" or "MTCS", default is "ICS" if an anisotropic angular distribution function is used and "MTCS" if an isotropic angular distribution function is used.
EP_0	(float) The initial energy (eV) of a hypothetical electron per time step, default is 10.
Ered	(float) Specify reduced electric field (Td), default is None.
NN	(int) Number of background neutral gas particles (assumed gas species is of index 1), default is None.
Nmin	(float) Minimum number of pseudo pairs to be generated by the smallest cross section of interest, default is 1.0.
analytic_cs	*(str or bool) For the built-in pytb.input.He_TB Helium model, use either tabulated data, analytic fits, or a mix of both, options are False | True | "mixed".
autostep	(bool) Flag to calculate DT / NP / E from Ered / L / pct_ion / DE / EP_0, default is False.
downsample	(bool) If specified with vdf_init, truncate the init file such that NP is satisfied, default is False.
duration	(float) Run simulation until duration (s) is complete, default is None.

`eadf`	(str) Elastic angular distribution function model — options are `"default"`, or `"He_Park"`.
`eesd`	(str) Electron energy sharing distribution model — options are `"default"` (one takes all) | `"equal"` | `"uniform"`.
`egen`	*(bool) Allow secondary electron generation for the ionization model.
`fixed_background`	(bool) Flag to append "FixedParticle2" to each of the reaction types in the indeck
`gas_index`	(int) The index of the neutral gas species (if applicable), default is `1`.
`indeck`	(callable or str) Function for auto-generating indeck and CS object or string path to directory with CS data and indeck, default is `None`.
`mix_thresh`	*(float) If `analytic_cs` is `"mixed"`, use numerical data at energies lower than this threshold value (in eV), and use analytic data at higher energies.
`n`	*(int) For the built-in `pytb.input.He_TB` Helium model, include CCC excitation processes from the ground state to (up to and including) states with principle quantum number `n`.
`nsamples`	*(int) For the built-in `pytb.input.He_TB`, this specifies the number of tabulated cross section values for analytic sampling.
`pc_scale`	(dict[str->float]) Multiply any property involved in the $N_{\min}$ constraint by a constant after the constraint is imposed; useful for convergence testing, default is `{}` (do nothing).
`pct_ion`	(float) Set the ratio $\frac{N_e}{N_{gas}}$, default is `None`.
`vdf_init`	(list[str or int or 2D-array, int]) Initialize particles with velocity data — a string at index `0` will read velocity data from that file path; an int at index `0` will attempt to reinitialize a previous calculation from a that time step if that step dump file is available; an array or DataFrame of shape (NP,3) at index `0` will create a velocity initialization file with the provided data, the value at index 1 should represent the species type.

Methods

get_params()	Return the set of parameters and their default values as a python dictionary.

pytb.parameters.WrapParameters.get_params

WrapParameters.**get_params**()

Return the set of parameters and their default values as a python dictionary.

4.1.2 Output Parameters

pytb.parameters.OutputParameters()	A listing of the main output parameters of the simulation, these keywords are the named columns of the time series and steady state data frames returned by *get_timeseries()* and *get_ss_params()* respectively.
pytb.parameters.ParticleParameters()	A listing of species dependent properties that can be accessed by *get_particle_tables()*, which returns a list of data tables (one for each species) where each column of data is labeled with one of the following keywords.

pytb.parameters.OutputParameters

class pytb.parameters.**OutputParameters**

A listing of the main output parameters of the simulation, these keywords are the named columns of the time series and steady state data frames returned by *get_timeseries()* and *get_ss_params()* respectively. These data tables also include the *ParticleParameters* of the species at index 0. The steady state parameters returned by *get_ss_params()* will also include standard deviations for each parameter indicated by an added "_std" suffix.

Attributes

`E`	(float) The electric field component (V/m) in the z direction, which can change in AC scenarios.
`MEe`	(float) The mean energy (eV) of the species at index 0 (usually electrons), computed as $\langle\epsilon\rangle = \frac{m_0}{2N_0}\sum_{i=1}^{N_0} v_{0i}^2$ where m_0 and N_0 are the mass and particle count of the 0^{th} species, and v_{0i} is the velocity vector of the i^{th} particle of species 0.
`a_n`	(float) The reduced flux Townshend ionization coefficient (m^2) of the species at index 0, computed as $\frac{\alpha^f}{n_{\text{gas}}} = \frac{1}{n_0 n_{\text{gas}}}\frac{dC_{\text{ion}}}{dt} \times \left(\frac{1}{N_0}\sum_{i=1}^{N_0} v_{\parallel,0i}\right)^{-1}$ where N_0 is the particle count of the 0^{th} species, C_{ion} is the count of ionization events, n_0 and n_{gas} are the 0^{th} and background gas densities repectively, and $v_{\parallel,0i}$ is velocity component parallel to the E field vector of the i^{th} th particle of species 0.
`a_n_bulk`	(float) The reduced bulk Townshend ionization coefficient (m^2) of the species at index 0, computed as $\frac{\alpha^b}{n_{\text{gas}}} = \frac{1}{n_0 n_{\text{gas}}}\frac{dC_{\text{ion}}}{dt} \times \left(\frac{d}{dt}\left[\frac{1}{N_0}\sum_{i=1}^{N_0} r_{\parallel,0i}\right]\right)^{-1}$, where N_0 is the particle count of the 0^{th} species, C_{ion} is the count of ionization events, n_0 and n_{gas} are the 0^{th} and background gas densities respectively, and $r_{\parallel,0i}$ is displacement component parallel to the E field vector of the i^{th} th particle of species 0.
`k_1`	(float) The rate coefficient for the first reaction, computed as $\frac{1}{n_0 n_{\text{gas}}}\frac{dC_1}{dt}$.
`k_ion`	(float) The ionization rate coefficient, computed as $\frac{1}{n_0 n_{\text{gas}}}\frac{dC_{\text{ion}}}{dt}$.
`mobN`	(float) The reduced flux mobility $(\text{V}^{-1}\text{m}^{-1}\text{s}^{-1})$ of the species at index 0, computed as $\mu^f n_{\text{gas}} = \frac{n_{\text{gas}}}{E}\frac{1}{N_0}\sum_{i=1}^{N_0} v_{\parallel,0i}$ where N_0 is the particle count of the 0^{th} species, E is field magnitude, n_{gas} is the density of the background gas, and $v_{\parallel,0i}$ is the velocity component parallel to the E field vector of the i^{th} th particle of species 0.
`mobN_bulk`	(float) The reduced bulk mobility $(\text{V}^{-1}\text{m}^{-1}\text{s}^{-1})$ of the species at index 0, computed as $\mu^b n_{\text{gas}} = \frac{n_{\text{gas}}}{E}\frac{d}{dt}\left[\frac{1}{N_0}\sum_{i=1}^{N_0} r_{\parallel,0i}\right]$ where N_0 is the particle count of the 0^{th} species, E is field magnitude, n_{gas} is the density of the background gas, and $r_{\parallel,0i}$ is displacement component parallel to the E field vector of the i^{th} th particle of species 0.
`n_gas`	(float) n_{gas}, the number density of the background gas.
`step`	(int) The number of time steps elapsed in the simulation, with `t` $= 0$ corresponding to `step` $= 0$, and with `t` = `DT` corresponding to `step` $= 1$.
`t`	(float) The time (s) elapsed in the simulation.

Methods

get_params()	Return the set of parameters and their default values as a python dictionary.

pytb.parameters.OutputParameters.get_params

OutputParameters.**get_params**()

Return the set of parameters and their default values as a python dictionary.

pytb.parameters.ParticleParameters

class pytb.parameters.**ParticleParameters**

A listing of species dependent properties that can be accessed by *get_particle_tables()*, which returns a list of data tables (one for each species) where each column of data is labeled with one of the following keywords.

Attributes

Ki	(float) The total kinetic energy (eV).
Mi	(float) The mean kinetic energy (eV).
Ni	(float) The number density (m $^{-3}$).
Rxi	(float) The mean x component of all particle displacements (m).
Ryi	(float) The mean y component of all particle displacements (m).
Rzi	(float) The mean z component of all particle displacements (m).
Txi	(float) The mean x component temperature (eV).
Tyi	(float) The mean y component temperature (eV).
Tzi	(float) The mean z component temperature (eV).
Vxi	(float) The mean x component velocity (m/s).
Vyi	(float) The mean y component velocity (m/s).
Vzi	(float) The mean z component velocity (m/s).
step	(int) The number of time steps elapsed in the simulation, with t $= 0$ corresponding to step $= 0$.
t	(float) The time (s) elapsed in the simulation.

Methods

get_params()	Return the set of parameters and their default values as a python dictionary.

pytb.parameters.ParticleParameters.get_params

ParticleParameters.**get_params**()

Return the set of parameters and their default values as a python dictionary.

4.1.3 Electron Growth and Memory Management

Depending on the ionization model and field strength, ThunderBoltz may generate a large number of electrons. In these cases, the appropriate amount of memory must be allocated. The correct amount will be allocated automatically in scenarios where no ionization process is used, or when the `IonizationNoEgen` model is used. This amount will be allocated based on the sum of all `NP` elements times 4.

However, in scenarios where there is significant electron generation, i.e. at high E fields with the `Ionization` model on, the default memory settings are not sufficient and the simulation will exit with the error "Too many particles!". To prevent this specify the `MEM` flag in the *ThunderBoltz* constructor:

```
tb = ThunderBoltz(
     # For example, using the Helium model.
     indeck=pytb.input.He_TB,
     # This will turn on electron generation for the Helium model
     # i.e. this will ensure the "Ionization" collision model is
     # used in the generated indeck.
     egen=True,
     # Now we must set the MEM flag, since we will be generating
     # a lot of electrons.
     MEM = 10, # in GB
)
```

MEM will accept any float representing the number of gigabytes to be made available to the particle arrays.

Warning: If the value of `MEM` is more than the actual number of available GB, then the simulation will still run, but will exit with a segmentation fault once too many particles are created.

Warning: When using multiple cores on the same machine / node, ensure that each process has enough memory requested and that the sum of memory requests does not exceed the available pool of RAM.

CHAPTER

FIVE

API REFERENCE

See the full *API Reference* for full documentation of the ThunderBoltz programming interface.

5.1 API Reference

ThunderBoltz([directory, cs, out_file, ...])	ThunderBoltz 0D DSMC simulation wrapper.

5.1.1 pytb.ThunderBoltz

class pytb.**ThunderBoltz**(*directory=None*, *cs=None*, *out_file=None*, *monitor=None*, *live=None*, *live_rate=None*, *ts_plot_params=None*, ***params*)

ThunderBoltz 0D DSMC simulation wrapper.

Parameters

- **directory** (*str*) – The path to a directory that will host ThunderBoltz compiled, source, input, and output files.
- **cs** (CrossSections) – The set of cross section information required for this simulation. Optionally supplied as an alternative to `indeck`, default is an empty CrossSections object.
- **out_file** (*str*) – Optional file base name for ThunderBoltz stdout buffer, default is `"thunderboltz"`.
- **monitor** (*bool*) – Runtime flag, when set to `True` an empty `monitor` file will be generated in the simulation directory. Deleting this file will cause the ThunderBoltz process to exit, but allow the wrapper to continue execution. This is useful performing several simulation calculations sequentially, but manual exit is required for each one, or if post processing is required immediately after ThunderBoltz exits.
- **live** (*bool*) – Run and update time series plotting GUI during simulation.
- **ts_plot_params** (*list[str]*) – The default output parameters to be plotted by *plot_timeseries()*.
- ****params** – pytb and ThunderBoltz simulation parameters. Any attributes of *TBParameters* or *WrapParameters* can be passed here.

Attributes

logfile	Name of the simulation output file produced by pytb
output_files	Files to be read when tabular data is requested
ts_plot_params	Time series plot parameters
particle_tables	Particle-specific times series data
kinetic_table	Banner output data
timeseries	All tick-by-tick simulation data
vdfs	Particle velocity dump data
vdf_init_data	Particle velocity data intended for particle initialization
time_conv	Time step at which steady-state calculations are considered converged
counts	Table of collision counts
elapsed_time	Elapsed wall-clock time of calculation
runtime_start	Date/Time of calculation start
runtime_end	Date/Time of calculation end
out_file	Name for ThunderBoltz stdout file
live	Run and update time series plotting GUI during simulation.
live_rate	Run and update reaction rate plotting GUI during simulation.
ts_fig	The figure object for the time series plot.
rate_fig	The figure object for the rate plot
callbacks	List of functions that are called every time banner output is updated.
directory	Simulation directory
err_stack	Recorded thunderboltz warnings read in from output files
monitor	Option to create temp file during run that causes safe exit upon deletion

Methods

add_callback(f)	Add a function to the list of functions that will be called during banner output.
compile_debug()	Prepare all files and compile with -g debug flag.
compile_from(src_path[, debug])	Copy TB files from src_path and compile in simulation directory.
describe_dist([steps, sample_cap])	Generate percentile and count statistics of the electron velocity / energy distribution for various time steps.
get_directory()	Return the path of the current simulation.
get_edfs([steps, sample_cap])	Read the electron velocity distribution functions and return the component and total energy distributions within a ThunderBoltz calculation.
get_etrans()	Return the energy weighted counts of each reaction computed for each time step.

get_particle_tables()	Return the particle table data for each species in a list.
get_sim_param(key)	Return the value of a simulation parameter.
get_ss_params([ss_func])	Get steady-state transport parameter values by averaging last section of time series.
get_timeseries()	Collect the relevant time series data from a ThunderBoltz simulation directory and add input parameter columns.
get_vdfs([steps, sample_cap, particle_type, v])	Read the electron velocities arrays within a ThunderBoltz calculation.
plot_cs([ax, legend, vsig, thresholds, save])	Plot the cross sections models.
plot_edf_comps([steps, sample_cap, bins, ...])	Plot the directional components of the energy distribution function.
plot_edfs([steps, sample_cap, bins, ...])	Plot the electron total energy distribution function, optionally include the provided cross sections for comparison.
plot_rates([save, stamp, v, update])	Create a diagnostic plot of ThunderBoltz time series data.
plot_timeseries([series, save, stamp, v, update])	Create a diagnostic plot of ThunderBoltz time series data.
plot_vdfs([steps, save, bins, sample_cap])	Plot the joint distribution heat map between the x-y and x-z velocities.
read([directory, read_input, read_cs_data, only])	Read the simulation directory of a ThunderBoltz run, possibly all of its input and output files.
read_log(logfile)	Read json file from simulation directory.
read_particle_table(i)	Read species specific output data, including density, velocity, displacement, energy, and temperature.
read_stdout(fname)	Read the banner output data.
read_tb_params(fname[, ignore])	Takes file name of an input deck, updates the simulation parameters and returns the simulation parameters which were read from the file `fname`.
reset()	Reset output data for a new run.
run([src_path, bin_path, out_file, monitor, ...])	Execute with the current parameters in the simulation directory.
*set_*(**p)	Update parameters, call appropriate functions ensuring input parameters are self-consistent.
set_fixed_tracking()	Change all reaction species indices of differing reactant values to be between only particle 0 and 1 (e.g.
set_ts_plot_params(params)	Set the default series plotted by *plot_timeseries()*
to_pickleable()	Return a picklable version of this object.
write_input(directory)	Write all the input files into a directory with the current settings.

pytb.ThunderBoltz.add_callback

ThunderBoltz.**add_callback**(*f*)

Add a function to the list of functions that will be called during banner output.

Parameters

f (*callable[,]*) – The function that will be called. It should accept no arguments and return no arguments.

pytb.ThunderBoltz.compile_debug

ThunderBoltz.**compile_debug**()

Prepare all files and compile with -g debug flag.

pytb.ThunderBoltz.compile_from

ThunderBoltz.**compile_from**(*src_path*, *debug=False*)

Copy TB files from src_path and compile in simulation directory.

Parameters

- **src_path** (*str*) – The location of the source files to compile from.
- **debug** (*bool*) – If `True`, compile C++ with the `-g` debug flag.

Raises

`RuntimeError` – if there is a compilation issue.

pytb.ThunderBoltz.describe_dist

ThunderBoltz.**describe_dist**(*steps='last'*, *sample_cap=500000*)

Generate percentile and count statistics of the electron velocity / energy distribution for various time steps.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - `"last"`: Only read the VDF of the last time step
  - `"first"`: Only read the VDF of the first time step
  - `"all"`: Read a separate VDF for each time step.
  - `list[int]`: Read VDF for each time step included in list.
  - `int`: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to `False`, then no cap will be imposed.

Returns

A table with statistical descriptions of the velocity and energy distributions.

Return type

`pandas.DataFrame`

pytb.ThunderBoltz.get_directory

ThunderBoltz.**get_directory**()

Return the path of the current simulation.

pytb.ThunderBoltz.get_edfs

ThunderBoltz.**get_edfs**(*steps='last'*, *sample_cap=500000*)

Read the electron velocity distribution functions and return the component and total energy distributions within a ThunderBoltz calculation. Energy units are in eV. Invokes *get_vdfs()*.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - "last": Only read the VDF of the last time step
  - "first": Only read the VDF of the first time step
  - "all": Read a separate VDF for each time step.
  - list[int]: Read VDF for each time step included in list.
  - int: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to False, then no cap will be imposed.

Returns

A table with the signed and unsigned
energy components of each particle.

Return type
pandas.DataFrame

pytb.ThunderBoltz.get_etrans

ThunderBoltz.**get_etrans**()

Return the energy weighted counts of each reaction computed for each time step.

Returns
A table of each process and the energy transfer through that channel as a proportion to the total energy transfer.

Return type
pandas.DataFrame

pytb.ThunderBoltz.get_particle_tables

ThunderBoltz.**get_particle_tables**()

Return the particle table data for each species in a list.

Returns
The list of particle table data. Each table with have columns with keywords matching the attributes of *ParticleParameters*.

Return type
list[pandas.DataFrame]

pytb.ThunderBoltz.get_sim_param

ThunderBoltz.**get_sim_param**(*key*)

Return the value of a simulation parameter.

Raises
IndexError – if the parameter is not set.

pytb.ThunderBoltz.get_ss_params

ThunderBoltz.**get_ss_params**(*ss_func=None*)

Get steady-state transport parameter values by averaging last section of time series. By default, the last fourth of the available data is considered to be steady-state. Standard deviations over this interval will be computed for each parameter in a new column with a "_std" suffix added to the column name.

Parameters
ss_func (callable[pandas.DataFrame, pandas.DataFrame]) – A function that takes in the numerical time series data and returns a truncated set of time series data that is considered to be at steady state.

Returns
The aggregated steady-state data for each output parameter along with columns specifying the input parameters.

Return type
pandas.DataFrame

Raises
RuntimeWarning – if not enough steps are available to compute steady state statistics.

Warning: Currently, the last quarter of the time series data is assumed to be in steady-state by default when calculating these steady-state parameters. One can verify that this is true by viewing the figures produced by *plot_timeseries()*. Otherwise, one may run the simulation for longer, or provide the appropriate criteria via ss_func.

pytb.ThunderBoltz.get_timeseries

ThunderBoltz.**get_timeseries**()

Collect the relevant time series data from a ThunderBoltz simulation directory and add input parameter columns.

Returns
The table of time series data.

Return type
pandas.DataFrame

pytb.ThunderBoltz.get_vdfs

ThunderBoltz.**get_vdfs**(*steps='last'*, *sample_cap=500000*, *particle_type=0*, *v=0*)

Read the electron velocities arrays within a ThunderBoltz calculation. Velocity units are in m/s. If velocity dump files are found corresponding to `steps`, update `vdfs`.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - `"last"`: Only read the VDF of the last time step
  - `"first"`: Only read the VDF of the first time step
  - `"all"`: Read a separate VDF for each time step.
  - `list[int]`: Read VDF for each time step included in list.
  - `int`: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to `False`, then no cap will be imposed.
- **particle_type** (*str, list[int], or int*) – Specify which kinds of species data should be read from.
  - `int`: The particle type to read. Default is `0`.
  - `list[int]`: A set of particle types to read.
  - `"all"`: Read all particle types.
- **v** (*int*) – Verbosity – 0: silent, 1: print file paths before reading.

Returns
The particle velocity dump data.

Return type
pandas.DataFrame

Warning: Large files are truncated to the first `sample_cap` lines. It is assumed that the particle ordering in the dump files is not correlated with any velocity statistics, but this may not be the case when `egen` is on. In that case, ensure the entire velocity dump file is being read.

pytb.ThunderBoltz.plot_cs

ThunderBoltz.**plot_cs**(*ax=None*, *legend=True*, *vsig=False*, *thresholds=False*, *save=None*, ***plot_args*)

Plot the cross sections models.

Parameters

- **ax** (Axes or None) – Optional axes object to plot on top of, default is None. If ax is None, then a new figure and ax object will be created.
- **legend** (*bool*) – Activate axes legend if true, default is True.
- **vsig** (*bool*) – Plot $\sqrt{\frac{2\epsilon}{m_e}}\sigma(\epsilon)$ rather than $\sigma(\epsilon)$.
- **thresholds** (*bool*) – if True, plot energy units in thresholds.
- **save** (*str*) – Optional location of the directory to save the plot in.
- ****plot_args** – Optional arguments passed to Axes.plot().

Returns

The axes object of the plot.

Return type

matplotlib.axes.Axes

pytb.ThunderBoltz.plot_edf_comps

ThunderBoltz.**plot_edf_comps**(*steps='last'*, *sample_cap=500000*, *bins=100*, *maxwellian=True*, *save=None*)

Plot the directional components of the energy distribution function.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - "last": Only read the VDF of the last time step
  - "first": Only read the VDF of the first time step
  - "all": Read a separate VDF for each time step.
  - list[int]: Read VDF for each time step included in list.
  - int: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to False, then no cap will be imposed.
- **bins** (*int*) – Total number of bins to divide the energy space into.
- **maxwellian** (*bool*) – Option to draw a maxwellian distribution with the same temperature for comparison.
- **save** (*str*) – Optional location of directory to save the figure in.

pytb.ThunderBoltz.plot_edfs

ThunderBoltz.**plot_edfs**(*steps='last'*, *sample_cap=500000*, *bins=100*, *plot_cs=False*, *save=None*)

Plot the electron total energy distribution function, optionally include the provided cross sections for comparison.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - `"last"`: Only read the VDF of the last time step
  - `"first"`: Only read the VDF of the first time step
  - `"all"`: Read a separate VDF for each time step.
  - `list[int]`: Read VDF for each time step included in list.
  - `int`: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to `False`, then no cap will be imposed.
- **bins** (*int*) – Total number of bins to divide the energy space into.
- **maxwellian** (*bool*) – Option to draw a maxwellian distribution with the same temperature for comparison.
- **save** (*bool*) – Optional location of directory to save the figure in.

Returns
The list of figures and a list of their corresponding step indices.

Return type
(Tuple[list[matplotlib.figure.Figure], list[int])

Note: It currently assumed that only data for one particle type is to be plotted.

pytb.ThunderBoltz.plot_rates

ThunderBoltz.**plot_rates**(*save=None*, *stamp=None*, *v=0*, *update=True*)

Create a diagnostic plot of ThunderBoltz time series data.

Parameters

- **series** (*list[str]*) – The y-parameters to plot onto the time series figure.
- **save** (*str*) – Option to save the plot to a file path.
- **stamp** (*list[str]*) – Option to stamp the figure with the value of descriptive parameters, e.g. the field, or initial number of particles. See *TBParameters* and *WrapParameters*.
- **v** (*int*) – Verbosity – 0: silent, 1: print file paths before plotting.
- **update** (*bool*) – If set to `False`, assume required data has already been parsed into ThunderBoltz frames.

Returns
The plot_rate figure object.

Return type
matplotlib.figure.Figure

pytb.ThunderBoltz.plot_timeseries

ThunderBoltz.**plot_timeseries**(*series=None*, *save=None*, *stamp=[]*, *v=0*, *update=True*)

Create a diagnostic plot of ThunderBoltz time series data.

Parameters

- **series** (*list[str]*) – The y-parameters to plot onto the time series figure.
- **save** (*str*) – Option to save the plot to a file path.
- **stamp** (*list[str]*) – Option to stamp the figure with the value of descriptive parameters, e.g. the field, or initial number of particles. See *TBParameters* and *WrapParameters*.
- **v** (*int*) – Verbosity – 0: silent, 1: print file paths before plotting.
- **update** (*bool*) – If set to `False`, assume required data has already been parsed into ThunderBoltz frames.

Returns
The timeseries figure object.

Return type
matplotlib.figure.Figure

pytb.ThunderBoltz.plot_vdfs

ThunderBoltz.**plot_vdfs**(*steps='last'*, *save=None*, *bins=100*, *sample_cap=500000*)

Plot the joint distribution heat map between the x-y and x-z velocities.

Parameters

- **steps** (*str, list[int], or int*) – Options for which time steps to read:
  - `"last"`: Only read the VDF of the last time step
  - `"first"`: Only read the VDF of the first time step
  - `"all"`: Read a separate VDF for each time step.
  - `list[int]`: Read VDF for each time step included in list.
  - `int`: read VDF at one specific time step.
- **sample_cap** (*int*) – Limit the number of samples read from the dump file for very large files. Default is 500000. If bool(sample_cap) evaluates to `False`, then no cap will be imposed.
- **bins** (*int*) – Total number of bins to divide the energy space into.
- **save** (*str*) – Optional location of directory to save the figure in.

Returns
The list of figures and a list of their corresponding step indices.

Return type
(Tuple[list[matplotlib.figure.Figure], list[int])

pytb.ThunderBoltz.read

ThunderBoltz.**read**(*directory=None*, *read_input=True*, *read_cs_data=False*, *only=None*)

Read the simulation directory of a ThunderBoltz run, possibly all of its input and output files.

Parameters

- **directory** (*str*) – The location of the simulation directory from which to read.
- **read_input** (*bool*) – Whether or not to read any input data.
- **read_cs_data** (*bool*) – Whether or not to read cross section data. This can be expensive, and often isn't necessary.
- **only** (*list or None*) – Only read certain types of files. Default is `["thunderboltz.out", "Particle_Type", "Counts.dat", ""thunderboltz.log"]`.

pytb.ThunderBoltz.read_log

ThunderBoltz.**read_log**(*logfile*)

Read json file from simulation directory. Update the corresponding settings in the ThunderBoltz object.

Parameters

logfile (*str*) – The path name of the logfile to read.

pytb.ThunderBoltz.read_particle_table

ThunderBoltz.**read_particle_table**(*i*)

Read species specific output data, including density, velocity, displacement, energy, and temperature.

Parameters

i (*int*) – The species index.

Returns

The particle data for species `i`.

Return type

pandas.DataFrame

pytb.ThunderBoltz.read_stdout

ThunderBoltz.**read_stdout**(*fname*)

Read the banner output data.

Parameters

fname (*str*) – The name of the `.out` file to read.

Returns

The banner data.

Return type

pandas.DataFrame

Note: If ThunderBoltz warnings are found (e.g. particle overload), a message will be appended to `err_stack`.

---

pytb.ThunderBoltz.read_tb_params

ThunderBoltz.**read_tb_params**(*fname*, *ignore=[]*)

Takes file name of an input deck, updates the simulation parameters and returns the simulation parameters which were read from the file `fname`.

Parameters

- **fname** (*str*) – The name of the indeck file to read.
- **ignore** (*list[str]*) – Don't read certain ThunderBoltz params, e.g. `["MP", "QP"]` would ignore the mass and charge parameters in an indeck file.

Returns
`tb_params`.

Return type
dict

pytb.ThunderBoltz.reset

ThunderBoltz.**reset**()

Reset output data for a new run.

pytb.ThunderBoltz.run

ThunderBoltz.**run**(*src_path=None*, *bin_path=None*, *out_file='thunderboltz'*, *monitor=False*, *dryrun=False*, *debug=False*, *std_banner=False*, *live=False*, *live_rate=False*)

Execute with the current parameters in the simulation directory.
The internal API ThunderBoltz version will be used in lieu of user-provided binary/source files.

Parameters

- **src_path** (*str*) – Optional path to source files to copy into the simulation directory. The source is then compiled there.
- **bin_path** (*str*) – Optional path to binary executable to copy into the simulation directory.
- **out_file** (*str*) – The file name for stdout buffer of the ThunderBoltz process.
- **monitor** (*bool*) – Runtime flag, when set to *True* will generate an empty *monitor* file in the simulation directory. Deleting this file will cause the ThunderBoltz process to exit, but allow the wrapper to continue execution. This is useful performing several simulation calculations sequentially, but a manual exit is required for each one.
- **dryrun** (*bool*) – Setup all the files for the calculation, but do not run the calculation.
- **debug** – (bool): Compile with C++ `-g` debug flag.
- **std_banner** (*bool*) – Toggle banner output streaming to stdout in addition to being written to the `out_file` buffer.
- **live** (*bool*) – Run and update time series plotting GUI during simulation.
- **live_rate** (*bool*) – Run and update rate plotting GUI during simulation.

Raises
`RuntimeError` – if there is no simulation directory set or if the one provided does not exist.

pytb.ThunderBoltz.set_

ThunderBoltz.**set_**(***p*)

Update parameters, call appropriate functions ensuring input parameters are self-consistent.

Parameters

****p** – Optional keyword parameters to update the calculator. can be any of *TBParameters* or *WrapParameters*.

pytb.ThunderBoltz.set_fixed_tracking

ThunderBoltz.**set_fixed_tracking**()

Change all reaction species indices of differing reactant values to be between only particle 0 and 1 (e.g. 0+1->0+2 is changed to 0+1->0+1).

pytb.ThunderBoltz.set_ts_plot_params

ThunderBoltz.**set_ts_plot_params**(*params*)

Set the default series plotted by *plot_timeseries()*

pytb.ThunderBoltz.to_pickleable

ThunderBoltz.**to_pickleable**()

Return a picklable version of this object.

pytb.ThunderBoltz.write_input

ThunderBoltz.**write_input**(*directory*)

Write all the input files into a directory with the current settings.

Parameters

directory (*str*) – The path to the simulation directory.

5.1.2 Build from Files

tb.read(directory[, read_cs_data])	Create a ThunderBoltz object by reading from a ThunderBoltz simulation directory.
tb.query_tree(directory[, name_req, ...])	Walk a directory tree and search for ThunderBoltz simulation directories to read.
tb.CrossSections.from_LXCat(fname)	Load cross section data from an LXCat .txt file

pytb.tb.read

pytb.tb.**read**(*directory*, *read_cs_data=False*)

Create a ThunderBoltz object by reading from a ThunderBoltz simulation directory.

Parameters

- **directory** (*str*) – The directory from which to initialize the ThunderBoltz object.
- **read_cs_data** (*bool*) – When set to true, the reader will look for cs_data, default is `False`

Returns

The ThunderBoltz object with tabulated data if available.

Return type

ThunderBoltz

pytb.tb.query_tree

pytb.tb.**query_tree**(*directory*, *name_req=None*, *param_req=None*, *read_cs_data=False*, *callback=None*, *agg=True*)

Walk a directory tree and search for ThunderBoltz simulation directories to read. Either return a list of *ThunderBoltz* objects, or a custom aggregation of the output data.

Parameters

- **directory** (*str*) – The root path to search for ThunderBoltz data in.
- **name_req** (*callable[str,bool]*) – A requirement on the file path names to be included in the query. The callable accepts the file path of a thunderboltz simulation directory and should return `True` if that directory is to be included in the query.

  e.g. `name_req=lambda s: "test_type_1" in s` would return only data in a subfolder `test_type_1`.
- **param_req** (*dict*) – A requirement on the parameter settings of the ThunderBoltz calculations. The dictionary corresponding to simulation parameters that must be set by the read ThunderBoltz object.

  e.g. `param_req={"Ered": 100, "L": 1e-6}` would only return data from calculations with a reduced field of 100 Td and a cell length of 1 μm.
- **callback** – (callable[*ThunderBoltz*, Any]): A function that accepts a ThunderBoltz object and returns the desired data.
- **agg** – If `callback` is set, attempt to aggregate the data based on the data type:

callback Return Type	Behavior
pandas.DataFrame	Frames will be concatenated row-wise and one larger DataFrame will be returned.
list[pandas.DataFrame]	A list of frames the same length of the return value will be returned. The frame at index `i` will contain the concatenated data from each simulation returned by `callable(tb)[i]`.
list[Any]	A list of lists will be returned. The list at index `i` will contain a list of items returned from each call to `callable(tb)[i]`.
Any	Return values will be returned in a list.

If `agg` is set to False, always return a list of callback data without any concatenation.

Returns
See `agg` option for behavior. Default return type is list[*ThunderBoltz*].

Return type
list[*ThunderBoltz*], or `pandas.DataFrame`, or list[`pandas.DataFrame`], or list[list[Any]]

pytb.tb.CrossSections.from_LXCat

`CrossSections.`**`from_LXCat`**`(`*fname*`)`

Load cross section data from an LXCat .txt file

5.1.3 Cross Sections

CrossSections([directory, input_path, ...])	ThunderBoltz cross section set data type.
Process(process_type[, r1, r2, p1, p2, ...])	A reaction process determined by reaction and product indices, a process type, a potential threshold, and a corresponding cross section specification.
input.He_TB([n, egen, analytic_cs, eadf, ...])	Generate parameterized He cross section sets in the ThunderBoltz format.
input.convert(df, u1, u2[, inv, drop, add])	Convert easily between units with a labeled DataFrame.

pytb.CrossSections

class pytb.**CrossSections**(*directory=None*, *input_path=None*, *read_cs_data=True*, *cs_dir_name='cross_sections'*, *input_fname=None*)

ThunderBoltz cross section set data type. Consists of a set of cross sections each with a file reference and a reaction table.

Parameters

- **directory** (*str*) – The path to a ThunderBoltz simulation directory in which input files are to be written. Default is `None`.
- **input_path** (*str*) – The path to a set of ThunderBoltz input files from which input data can be read. The file structure should be something like:

```
path/to/input_path
 |---indeck_file.in  <- The main ThunderBoltz indeck file.
 |---cross_sections  <- Cross section directory
 |   |---cs1.dat     <- ThunderBoltz-formatted cross section file.
 |   |---cs2.dat     <- ThunderBoltz-formatted cross section file.
 |   ...
 ...
```

- **cs_dir_name** (*str*) – The name of the cross section directory.
- **input_fname** (*str*) – The name of the main ThunderBoltz indeck file. Default is None, in which case the indeck will be searched for in *input_path* and must end with *.in*.

Attributes

cs_dir_name	Place for cs files in simulation dir
input_fname	Input deck filename default
table	The reaction table, with columns
data	Data tables for each cross section.
input_path	Input path to default input data

Methods

add_differential_model(rtype, name[, params])	Add a differential model to a certain type of process.
add_process(p)	Take a Process object and update the cross section data and cross section reaction table.
add_processes(ps)	Add multiple cross sections to the reaction table.
find_infile()	Look for indeck file in input_path.
from_LXCat(fname)	Load cross section data from an LXCat .txt file
get_deck()	Return the string formatted cross section table portion of the ThunderBoltz indeck.
plot_cs([ax, legend, vsig, thresholds])	Plot the cross sections models.
read(input_path[, read_cs_data])	Read ThunderBoltz cross section data from a directory with a single input file and a set of cross section files.
set_fixed_background([fixed])	Set all particle conserving processes to have the *FixedParticle2* tag or not.
write([directory])	Write cross section files into the simulation cross section directory.

pytb.CrossSections.add_differential_model

CrossSections.**add_differential_model**(*rtype*, *name*, *params=None*)

Add a differential model to a certain type of process.

Parameters

- **rtype** (*str*) – "Elastic", "Inelastic", or "Ionization", the broad collision process type.
- **name** (*str*) – The name of the differential process model. Available built-in options for each rtype are:

rtype	name
Elastic	Park, Murphy
Ionization	Equal, Uniform

- **params** (*list[float]*) – Optional list of parameters required by the differential model.

pytb.CrossSections.add_process

CrossSections.**add_process**(*p*)

Take a Process object and update the cross section data and cross section reaction table.

Parameters

p (Process) – The process object for a single type of interaction.

pytb.CrossSections.add_processes

CrossSections.**add_processes**(*ps*)

Add multiple cross sections to the reaction table.

Parameters

ps (*list[Process]*) – A list of process objects to add.

pytb.CrossSections.find_infile

CrossSections.**find_infile**()

Look for indeck file in input_path.

Raises

- **RuntimeError** – If input_path is not set, if multiple indecks
- **are found, or if no indecks are found.** –

pytb.CrossSections.from_LXCat

CrossSections.**from_LXCat**(*fname*)

Load cross section data from an LXCat .txt file

pytb.CrossSections.get_deck

CrossSections.**get_deck**()

Return the string formatted cross section table portion of the ThunderBoltz indeck.

pytb.CrossSections.plot_cs

CrossSections.**plot_cs**(*ax=None*, *legend=True*, *vsig=False*, *thresholds=False*, ***plot_args*)

Plot the cross sections models.

Parameters

- **ax** (Axes or None) – Optional axes object to plot on top of, default is None. If ax is None, then a new figure and Axes object will be created.
- **legend** (*bool or dict*) – Activate axes legend if true, default is True. If a dictionary is passed, it is interpreted as arguments to legend().
- **vsig** (*bool*) – Plot $\sqrt{\frac{2\epsilon}{m_e}}\sigma(\epsilon)$ rather than $\sigma(\epsilon)$ on the y-axis.

- **thresholds** (*bool*) – if `True`, plot $\frac{\epsilon}{\epsilon_{\text{ion}}} - 1$ rather than ϵ on the x-axis.
- ****plot_args** – Optional arguments passed to Axes.plot().

Returns
The axes object.

Return type
ax (matplotlib.axes.Axes)

pytb.CrossSections.read

`CrossSections.`**read**(*input_path*, *read_cs_data=True*)

Read ThunderBoltz cross section data from a directory with a single input file and a set of cross section files.

Parameters

- **input_path** (*str*) – The path to the directory with cross section data. If specified, CrossSections.input_path will be updated as well.
- **read_cs_data** (*bool*) – If `False`, only read the process header information, and not the actual cs data itself, default is `True`.

pytb.CrossSections.set_fixed_background

`CrossSections.`**set_fixed_background**(*fixed=True*)

Set all particle conserving processes to have the *FixedParticle2* tag or not.

pytb.CrossSections.write

`CrossSections.`**write**(*directory=None*)

Write cross section files into the simulation cross section directory.

Parameters
directory (*str*) – Option to write to a specific directory. If provided, `CrossSections.cs_dir` will be updated. Default is None.

Raises
`RuntimeError` – If no directory is set or provided.

pytb.Process

class `pytb.`**Process**(*process_type*, *r1=0*, *r2=1*, *p1=0*, *p2=1*, *threshold=0.0*, *cs_func=None*, *cs_data=None*, *name=None*, *differential_process=None*, *nsamples=250*)

A reaction process determined by reaction and product indices, a process type, a potential threshold, and a corresponding cross section specification.

process_type: str
Elastic | Inelastic | Ionization

r1,r2,p1,p2: int (0,1,0,1)
The indices of the reactants and products.

threshold: float
The threshold value for the process (e.g. the binding energy of an ionization process).

cs_func: callable
Function that returns the cross section for this process in m^2 given an incident electron energy in eV (center of mass frame).

cs_data: 2-D Array-Like
Tabular cross section data with columns of energy (eV) and cross section (m^2).

Attributes

SAMPLE_MAX
SAMPLE_MIN
cs_func
nsamples

Methods

add_differential_parameters(name, params)	Typically differential processes require analytic forms due to the difficulty of extrapolation in several dimensions.
auto_sample()	Check if the cross section needs a dense grid or not by comparing simple and dense grids.
require_cs()	Ensure there is some kind of cross section data associated with this process.
sample_cs([e_points, grid_type, nsamples])	Sample self.cs_func on a grid of energies.
to_cs_frame(a)	Convert any kind of two-dimensional data to a pandas DataFrame with columns *Energy (eV)* and *Cross Section (m^2)*
to_df()	Convert to properly formatted pandas DataFrame.
zero_below_thresh()	Enforce the ThunderBoltz required cross section format.

pytb.Process.add_differential_parameters

Process.**add_differential_parameters**(*name*, *params*)

Typically differential processes require analytic forms due to the difficulty of extrapolation in several dimensions. Add free parameters into an analytic differential model here.

Parameters

- **name** – The name of the model for this differential process.
- **params** – The free parameters required for this differential model.

pytb.Process.auto_sample

Process.**auto_sample**()

Check if the cross section needs a dense grid or not by comparing simple and dense grids.

pytb.Process.require_cs

Process.**require_cs**()

Ensure there is some kind of cross section data associated with this process.

Raises

RuntimeError – if there is no cross section data available.

pytb.Process.sample_cs

Process.**sample_cs**(*e_points=None*, *grid_type='log dense'*, *nsamples=None*)

Sample self.cs_func on a grid of energies.

Parameters

- **e_points** (*ArrayLike*) – Explicit energy (eV) grid points on which to sample.
- **grid_type** (*str*) –

"log dense"	sample nsamples near threshold up to 1MeV.
"simple"	sample 0 eV - threshold - 1 MeV
None	Behavior will automatically be determined by auto_sample.

- **nsamples** (*int*) – Override self.nsamples for this sampling call.

pytb.Process.to_cs_frame

Process.**to_cs_frame**(*a*)

Convert any kind of two-dimensional data to a pandas DataFrame with columns *Energy (eV)* and *Cross Section (m^2)*

pytb.Process.to_df

Process.**to_df**()

Convert to properly formatted pandas DataFrame.

Returns

The process information.

Return type

(pandas.DataFrame)

Raises

RuntimeError – if no data is available to produce a DataFrame

pytb.Process.zero_below_thresh

Process.**zero_below_thresh**()

Enforce the ThunderBoltz required cross section format. For processes with a non-zero threshold, include zero valued points at 0 eV and at threshold energy.

Raises
RuntimeError – if there is no cross section data to format.

pytb.input.He_TB

pytb.input.**He_TB**(*n=4*, *egen=True*, *analytic_cs=True*, *eadf='default'*, *ECS=None*, *nsamples=250*, *mix_thresh=300.0*, *fixed_background=True*)

Generate parameterized He cross section sets in the ThunderBoltz format. The data is from Igor Bray and Dmitry V Fursa 2011 J. Phys. B: At. Mol. Opt. Phys. 44 061001.

Parameters

- **n** (*int*) – Include CCC excitation processes from the ground state to (up to and including) states with principle quantum number `n`.
- **egen** (*bool*) – Allow secondary electron generation for the ionization model.
- **analytic_cs** (*str or bool*) – use either tabulated data, analytic fits, or a mix of both. Options are `False`, `True`, or `"mixed"`.
- **eadf** (*str*) – `"default"`, or `"He_Park"`.
- **ECS** (*str or None*) – The total elastic cross section model. Options are `"ICS"` or `"MTCS"`, default is `"ICS"` if an anisotropic angular distribution function is used and `"MTCS"` if an isotropic angular distribution function is used.
- **nsamples** (*int*) – The number of tabulated cross section values for analytic sampling.
- **mix_thresh** (*float*) – If `analytic_cs` is `"mixed"`, use numerical data at energies lower than this threshold value (in eV), and use analytic data at higher energies.
- **fixed_background** (*bool*) – Flag to append "FixedParticle2" to each of the reaction types in the indeck.

Returns

The *CrossSections* object for Helium
and the dictionary of ThunderBoltz parameters suitable for the cross section model.

Return type
Tuple[dict,dict]

pytb.input.convert

pytb.input.**convert**(*df*, *u1*, *u2*, *inv=False*, *drop=False*, *add=False*)

Convert easily between units with a labeled DataFrame. Columns in the format *<name> (<unit>)* will be converted.

5.1.4 Parallel Computing

parallel.MPRunner()	Interface for any kind of calculation that is run-able and set-able can be compatible with multiprocessing utilities like SlurmManager and DistributedPool.
parallel.DistributedPool(runner[, processes])	A multiprocessing Pool context for running calculations among cores with different settings.
parallel.SlurmManager(runner[, directory, ...])	A python context interface for the common Slurm HPC job manager to run more several intensive calculations on large clusters.

pytb.parallel.MPRunner

class pytb.parallel.**MPRunner**

Interface for any kind of calculation that is run-able and set-able can be compatible with multiprocessing utilities like SlurmManager and DistributedPool.

Methods

get_directory()	Return the directory in which the calculation is occurring.
run(**run_options)	Run the calculation with the current settings.
*set_*(**state_options)	Update the internal state of the object being run.
to_pickleable()	Returns a pickle-able portion of the object sufficient to run the calculations.

pytb.parallel.MPRunner.get_directory

MPRunner.**get_directory**()

Return the directory in which the calculation is occurring.

Returns

(str): The path of the directory in which the program is being run.

pytb.parallel.MPRunner.run

MPRunner.**run**(***run_options*)

Run the calculation with the current settings.

Parameters

****run_options** – Keywords arguments that modify the nature of the way the program runs.

pytb.parallel.MPRunner.set_

MPRunner.**set_**(***state_options*)

Update the internal state of the object being run.

Parameters

****state_options** – Keywords arguments corresponding to attributes of the object being updated.

pytb.parallel.MPRunner.to_pickleable

MPRunner.**to_pickleable**()

Returns a pickle-able portion of the object sufficient to run the calculations.

pytb.parallel.DistributedPool

class pytb.parallel.**DistributedPool**(*runner:* MPRunner, *processes=None*)

A multiprocessing Pool context for running calculations among cores with different settings.

Parameters

- **runner** (MPRunner) – The calculation runner.
- **processes** (*int*) – The number of cores to divide up the work.

Methods

err_callback(err)	Print out errors that subprocesses encounter.
submit([run_args])	Submit a single job with updated key words to the pool.

pytb.parallel.DistributedPool.err_callback

DistributedPool.**err_callback**(*err*)

Print out errors that subprocesses encounter.

pytb.parallel.DistributedPool.submit

DistributedPool.**submit**(*run_args={}*, ***set_args*)

Submit a single job with updated key words to the pool.

Parameters

run_args (*dict*) – Keyword arguments to be passed to *run()*.

****set_args: Keyword arguments passed to**
set_() before calling *run()*.

pytb.parallel.SlurmManager

class pytb.parallel.**SlurmManager**(*runner:* MPRunner, *directory=None*, *modules=['python', 'gcc']*, *mock=False*, ***options*)

A python context interface for the common Slurm HPC job manager to run more several intensive calculations on large clusters. See https://slurm.schedmd.com/sbatch.html.

Parameters

- **runner** (*MPRunner*) – The calculation runner.
- **directory** (*str or None*) – the current working directory.
- **modules** (*list[str]*) – A list of modules to be loaded by the HPC module system.
- **mock** (*bool*) – Option to test scripts without calling a slurm manager.
- ****options** – Additional keyword arguments will be interpreted as SLURM parameters.

Note: This job manager currently only works for clusters that either already have the gcc and python requirements installed on each compute node, or clusters that use the Module System to load functionality.

The default behavior is to accommodate the module system as it is common on most HPC machines. If you wish to avoid writing `module load` commands in the SLURM script, simply specify `modules=[]` in the constructor.

Attributes

directory	The simulation directory
modules	The list of modules to be loaded by the HPC module system.
runner	The *MPRunner* object.
job_ids	Store references to the slurm job numbers after jobs are submitted
options	The SLURM sbatch options

Methods

batch_script()	The SLURM job script.
has_active()	Check whether any submitted jobs are still pending or running.
has_pending()	Check whether any submitted jobs are still pending.
join()	Wait for all slurm jobs to finish.
mock_run()	Act as a compute node and test the job scripts sequentially.
process_batch_script()	Inspect the batch script below and process it for use in sbatch.
sbatch()	Call slurm with current settings.
*set_*(**options)	Update slurm manager options.
submit([run_args])	Add a set of parameter updates to the job queue.
write_slurm_script([path, script_name])	Write the SLURM batch script.

pytb.parallel.SlurmManager.batch_script

SlurmManager.**batch_script**()

The SLURM job script. This does not get called in the parent process, but instead the source code is invoked in the sbatch script/command for subprocess startup.

pytb.parallel.SlurmManager.has_active

SlurmManager.**has_active**()

Check whether any submitted jobs are still pending or running.

Returns

True if there are still jobs that are pending or
running. `False` otherwise.

Return type
(bool)

pytb.parallel.SlurmManager.has_pending

SlurmManager.**has_pending**()

Check whether any submitted jobs are still pending.

Returns

True if there are still jobs that are pending.
`False` otherwise.

Return type
(bool)

pytb.parallel.SlurmManager.join

SlurmManager.**join**()

Wait for all slurm jobs to finish.

pytb.parallel.SlurmManager.mock_run

SlurmManager.**mock_run**()

Act as a compute node and test the job scripts sequentially.

pytb.parallel.SlurmManager.process_batch_script

SlurmManager.**process_batch_script**()

Inspect the batch script below and process it for use in sbatch.

pytb.parallel.SlurmManager.sbatch

SlurmManager.**sbatch**()

Call slurm with current settings.

pytb.parallel.SlurmManager.set_

SlurmManager.**set_**(***options*)

Update slurm manager options.

Parameters

****options** – SLURM settings.

pytb.parallel.SlurmManager.submit

SlurmManager.**submit**(*run_args={}*, ***settings*)

Add a set of parameter updates to the job queue. Slurm is not invoked until the context is exited.

Parameters

- **run_args** (*dict*) – Keyword arguments to be passed to *run()*.
- ****settings** – Keyword arguments passed to the *set_()*. before calling *run()*.

pytb.parallel.SlurmManager.write_slurm_script

SlurmManager.**write_slurm_script**(*path=None*, *script_name=None*)

Write the SLURM batch script.

INDEX

Q

R

S

T

W

Z