

---

# **ThunderBoltz API Manual**

Ryan Park, Brett Scheiner, Mark Zammit

Los Alamos National Laboratory, Los Alamos, NM, 87545

Jun 21, 2024  
LA-UR-23-32356

# CONTENTS

<b>1</b>	<b>Installation</b>	<b>2</b>
<b>2</b>	<b>Tutorials</b>	<b>3</b>
2.1	Quick Start Guide . . . . .	3
2.2	Preparing Cross Sections . . . . .	9
2.3	Running Multiple Calculations . . . . .	12
2.4	Extracting Results . . . . .	14
<b>3</b>	<b>Benchmark Testing</b>	<b>18</b>
3.1	Benchmark Testing . . . . .	18
<b>4</b>	<b>Simulation Parameters</b>	<b>20</b>
4.1	Simulation Parameters . . . . .	20
<b>5</b>	<b>API Reference</b>	<b>30</b>
5.1	API Reference . . . . .	30
	<b>Index</b>	<b>57</b>

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

## INSTALLATION

The Python API can be installed via *pip*.

```
pip install thunderboltz
```

The `thunderboltz` package ships with the C++ ThunderBoltz source and requires only a gcc compiler to run complete calculations.

The code can also be downloaded from the repository. Use the following command to clone the code into a local repository.

```
git clone git@github.com:lanl/ThunderBoltz.git
```

You may need to set up SSH keys in order to access Github. See the [Github SSH Guide](#) to set up access to Github 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/cpp`. To install the Python interface, run the `install.sh` script from the root directory.

```
./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.

## 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 thunderboltz import ThunderBoltz # Import the main simulation object
from thunderboltz.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")

calc = 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.
calc.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 = calc.get_timeseries()
# This will extract the same quantities but time averaged during the
# steady state period of the run.
steady_state = calc.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 thunderboltz import read

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

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

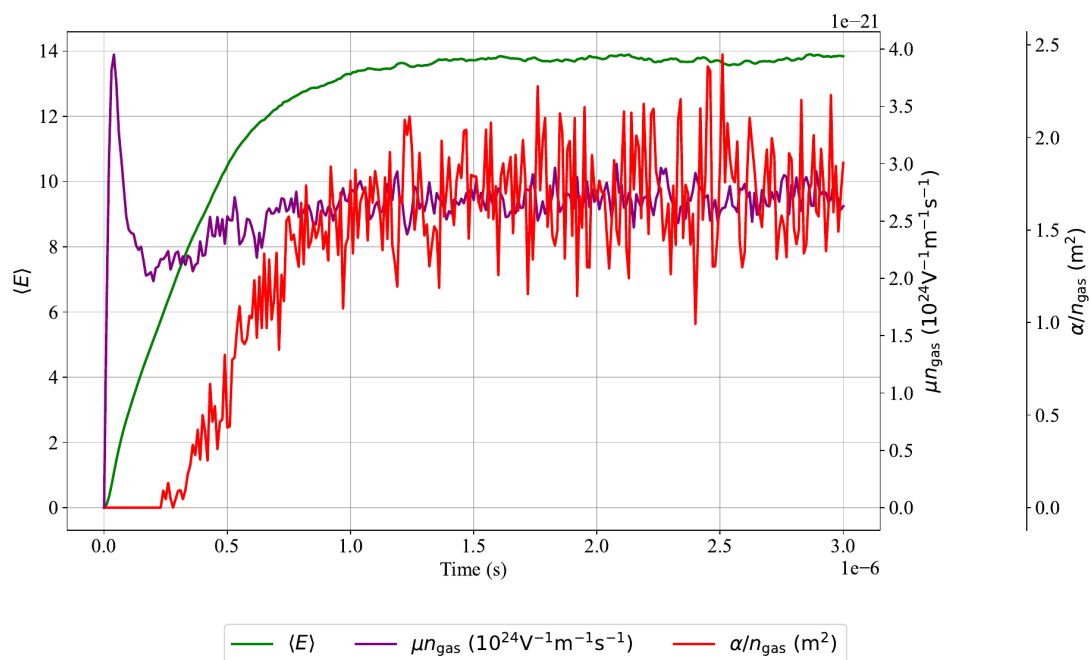
# One can truncate the data row-wise. For example,
# the following will take data from last 20000 steps
# of the 30000 steps.
trunc = timeseries[timeseries.step >= 10000].copy()

# 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()
```

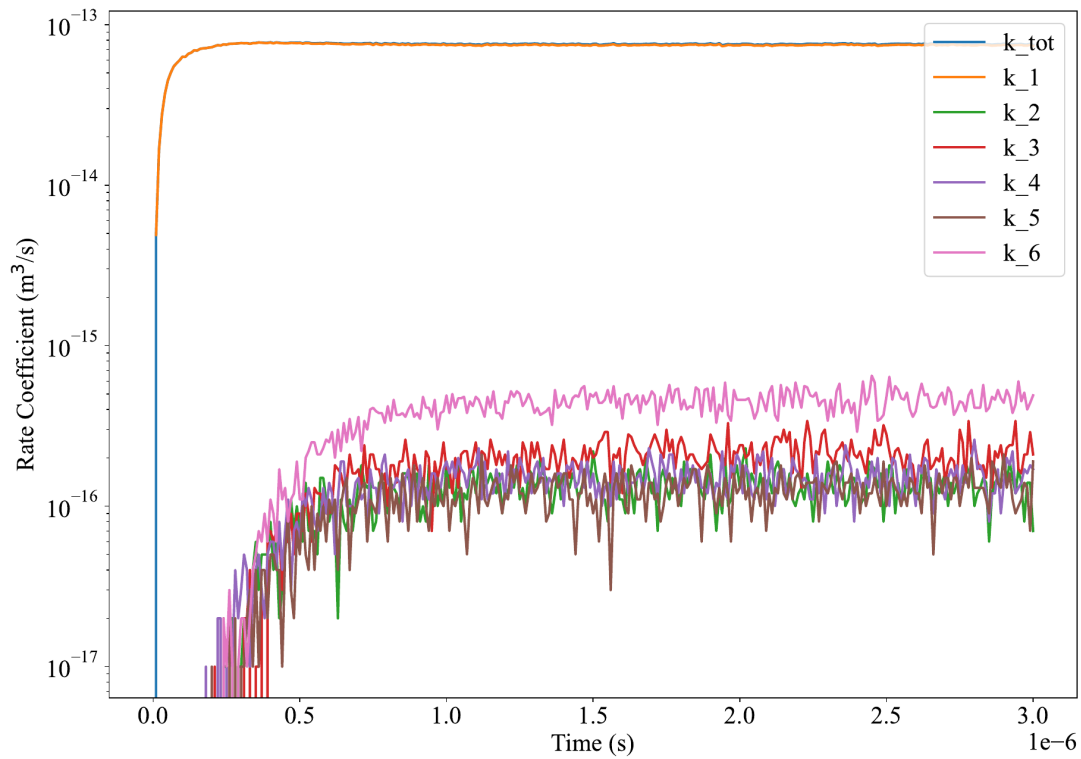
There are also some built-in plotting methods that can be accessed through the ThunderBoltz object.

```
# 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
calc.plot_timeseries()
```

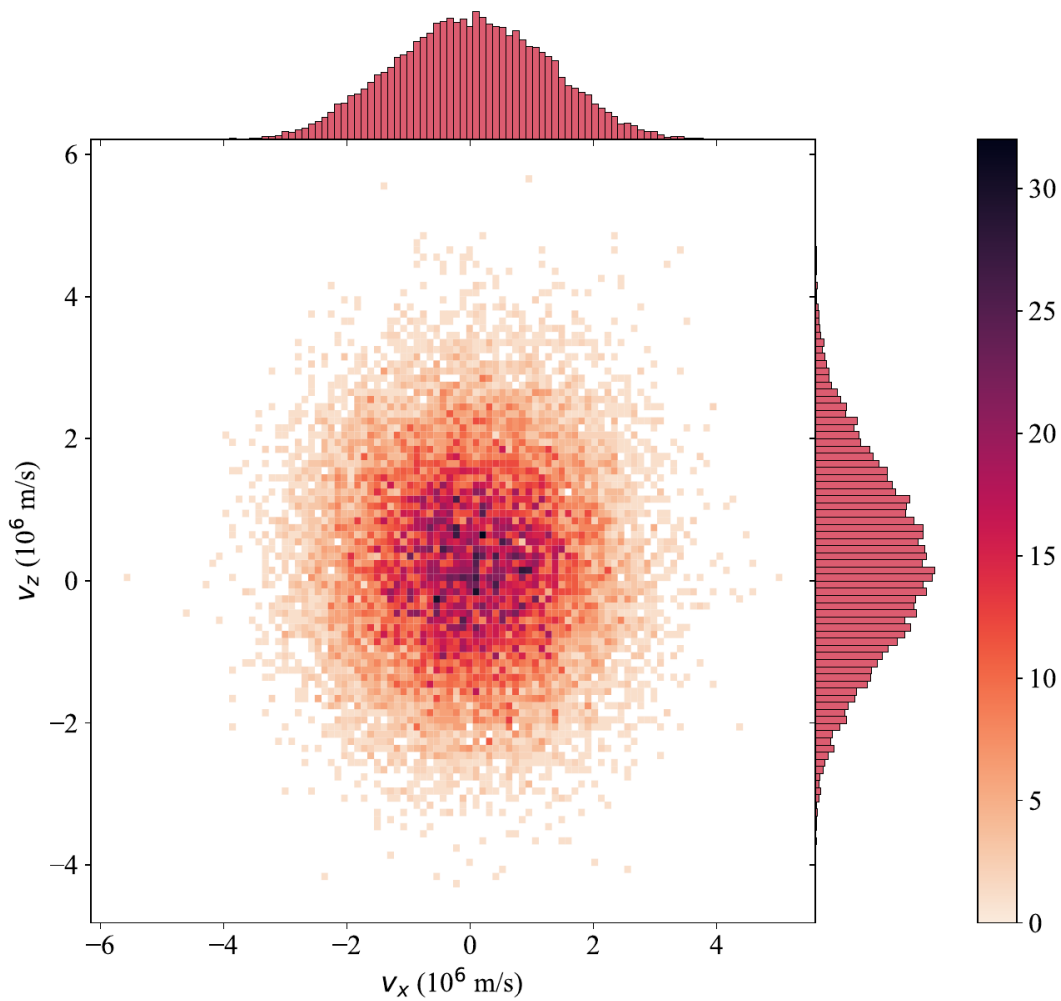


```
# This will plot the rate coefficients for every process.
calc.plot_rates()
```





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



In order to actually view plots in a GUI, make sure to call the matplotlib function `show`,

```
# 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 thunderboltz 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*.

2. By reading from an LXCat text file extract.

```
from thunderboltz 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.

3. By programmatically generating cross section data in python.

This approach involves the *Process* object.

```
from thunderboltz import CrossSections
from thunderboltz 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],
```

(continues on next page)

(continued from previous page)

```

    [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
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)

```

(continues on next page)

(continued from previous page)

```
# 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.

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

calc.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 thunderboltz as tb

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

calc = ThunderBoltz(indeck=tb.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)
    calc.set_(Ered=field, directory=subdir)
    # Run the calculation
    calc.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 thunderboltz as tb

# 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
calc = ThunderBoltz(indeck=tb.input.He_TB)

fields = [10, 100, 500]

# This time use the DistributedPool context,
# passing the ThunderBoltz object like so
with DistributedPool(calc) as pool:
    # Loop through the field values
```

(continues on next page)

(continued from previous page)

```

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 thunderboltz as tb

# 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
calc = ThunderBoltz(indeck=tb.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,

```

(continues on next page)

(continued from previous page)

```
# but also give it your SLURM options.
with SlurmManager(calc, 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 thunderboltz as tb

# Pass the location of the simulation directory to be read
calc = tb.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 thunderboltz as tb

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

# You will now see the cross section data has been loaded
print(calc.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 thunderboltz as tb

calcs = tb.query_tree("path/to/base_path")

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

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

# Plot the last velocity dump data from the fourth read calculation.
calcs[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 thunderboltz as tb

calcs = tb.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(calcs[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

<code>thunderboltz.ThunderBoltz.plot_timeseries([...])</code>	Create a diagnostic plot of ThunderBoltz time series data.
<code>thunderboltz.ThunderBoltz.plot_rates([save, ...])</code>	Create a diagnostic plot of ThunderBoltz time series data.
<code>thunderboltz.ThunderBoltz.plot_edf_comps([...])</code>	Plot the directional components of the energy distribution function.
<code>thunderboltz.ThunderBoltz.plot_edfs([steps, ...])</code>	Plot the electron total energy distribution function, optionally include the provided cross sections for comparison.
<code>thunderboltz.ThunderBoltz.plot_cs([ax, ...])</code>	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 thunderboltz as tb

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

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

# Call plotting methods
calc.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. For example:

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

## BENCHMARK TESTING

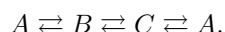
See *Benchmark Testing* to run code that reproduces the results found in the paper.

### 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<sup>1</sup> predicts the kinetic rates of the following chemical reactions between arbitrary heavy particles,



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

$$k(T) = d^2 \sqrt{\frac{8\pi k_B T}{m_r}} e^{-E_a/k_B T}.$$

To run this system 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.

---

<sup>1</sup> 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<sup>2</sup>, run the prepared function either in a python script or 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,  $\mu N$ , and Townshend ionization coefficient,  $\alpha/N$ , 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()"
```

---

<sup>2</sup> K F Ness 1994 J. Phys. D: Appl. Phys. 27 1848.

## 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

<code>thunderboltz.parameters.TBParameters()</code>	The ThunderBoltz simulation settings and their default values.
<code>thunderboltz.parameters.WrapParameters()</code>	Additional Python interface settings and their defaults.

#### **thunderboltz.parameters.TBParameters**

**class** thunderboltz.parameters.TBParameters

The ThunderBoltz simulation settings and their default values.

## Attributes

B	(list[int]) Magnetic field vector (Tesla), default is $[0, 0, 0]$ .
CR	(int) If 1, then the remainder of $N_{\text{pairs}}$ is carried into the next $N_{\text{pairs}}$ 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 $-24640$ .
ET	(float) E-field oscillation frequency (Hz), default is 0.
EX	(int) Cross section extrapolation — options are 0 (extrapolated cross sections are set to $0 \text{ m}^2$ ), 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 $[5.4857e-4, 28]$ .
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 $[-1, 0]$ .
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 $[0, 0.0259]$ .
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 $[0, 0]$ .

## Methods

<code>get_params()</code>	Return the set of parameters and their default values as a python dictionary.
---------------------------	---

### `thunderboltz.parameters.TBParameters.get_params`

`TBParameters.get_params()`

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

### `thunderboltz.parameters.WrapParameters`

**class** `thunderboltz.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

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



<code>eadf</code>	(str) Elastic angular distribution function model — options are "default", or "He_Park".
<code>eesd</code>	(str) Electron energy sharing distribution model — options are "default" (one takes all)   "equal"   "uniform".
<code>egen</code>	*(bool) Allow secondary electron generation for the ionization model.
<code>fixed_background</code>	(bool) Flag to append "FixedParticle2" to each of the reaction types in the indeck
<code>gas_index</code>	(int) The index of the neutral gas species (if applicable), default is 1.
<code>indeck</code>	(callable or str) Function for auto-generating indeck and CS object or string path to directory with CS data and indeck, default is None.
<code>mix_thresh</code>	*(float) If <code>analytic_cs</code> is "mixed", use numerical data at energies lower than this threshold value (in eV), and use analytic data at higher energies.
<code>n</code>	*(int) For the built-in <code>thunderboltz.input.He_TB</code> Helium model, include CCC excitation processes from the ground state to (up to and including) states with principle quantum number <code>n</code> .
<code>nsamples</code>	*(int) For the built-in <code>thunderboltz.input.He_TB</code> , this specifies the number of tabulated cross section values for analytic sampling.
<code>pc_scale</code>	(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).
<code>pct_ion</code>	(float) Set the ratio $\frac{N_e}{N_{\text{gas}}}$ , default is None.
<code>vdf_init</code>	(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

<code>get_params()</code>	Return the set of parameters and their default values as a python dictionary.
---------------------------	---

**thunderboltz.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**

<code>thunderboltz.parameters.OutputParameters()</code>	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 <code>get_timeseries()</code> and <code>get_ss_params()</code> respectively.
<code>thunderboltz.parameters.ParticleParameters()</code>	A listing of species dependent properties that can be accessed by <code>get_particle_tables()</code> , which returns a list of data tables (one for each species) where each column of data is labeled with one of the following keywords.

**thunderboltz.parameters.OutputParameters****class thunderboltz.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

D_H	(float) The hall components of the flux diffusion tensor $\overleftrightarrow{D}^f = \langle \mathbf{r}\mathbf{v} \rangle - \langle \mathbf{r} \rangle \langle \mathbf{v} \rangle$ are combined into the flux Hall diffusion $D_H^f = D_{xz}^f + D_{zx}^f$ .
D_H_bulk	(float) The hall components of the bulk diffusion tensor $\overleftrightarrow{D}^b = \frac{1}{2} \frac{d}{dt} \langle (\mathbf{r} - \langle \mathbf{r} \rangle)^2 \rangle$ are combined into the bulk Hall diffusion $D_H^b = D_{xz}^b + D_{zx}^b$ .
D_XX	(float) The diagonal components of the flux diffusion tensor $\overleftrightarrow{D}^f = \langle \mathbf{r}\mathbf{v} \rangle - \langle \mathbf{r} \rangle \langle \mathbf{v} \rangle$ are output under D_XX, D_YY, and D_ZZ.
D_XX_bulk	(float) The diagonal components of the bulk diffusion tensor $\overleftrightarrow{D}^b = \frac{1}{2} \frac{d}{dt} \langle (\mathbf{r} - \langle \mathbf{r} \rangle)^2 \rangle$ are output under D_XX_bulk, D_YY_bulk, and D_ZZ_bulk.
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 <sup>th</sup> species, and $v_{0i}$ is the velocity vector of the $i^{\text{th}}$ particle of species 0.
a_n	(float) The reduced flux Townshend ionization coefficient ( $\text{m}^2$ ) 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 <sup>th</sup> species, $C_{\text{ion}}$ is the count of ionization events, $n_0$ and $n_{\text{gas}}$ are the 0 <sup>th</sup> and background gas densities respectively, and $v_{\parallel,0i}$ is velocity component parallel to the E field vector of the $i^{\text{th}}$ particle of species 0.
a_n_bulk	(float) The reduced bulk Townshend ionization coefficient ( $\text{m}^2$ ) 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 <sup>th</sup> species, $C_{\text{ion}}$ is the count of ionization events, $n_0$ and $n_{\text{gas}}$ are the 0 <sup>th</sup> and background gas densities respectively, and $r_{\parallel,0i}$ is displacement component parallel to the E field vector of the $i^{\text{th}}$ 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 ( $V^{-1}m^{-1}s^{-1}$ ) of the species at index 0, computed as $\mu^f n_{gas} = \frac{n_{gas}}{E} \frac{1}{N_0} \sum_{i=1}^{N_0} v_{  ,0i}$ where $N_0$ is the particle count of the 0 <sup>th</sup> species, $E$ is field magnitude, $n_{gas}$ is the density of the background gas, and $v_{  ,0i}$ is the velocity component parallel to the E field vector of the $i^{th}$ particle of species 0.
mobN_bulk	(float) The reduced bulk mobility ( $V^{-1}m^{-1}s^{-1}$ ) of the species at index 0, computed as $\mu^b n_{gas} = \frac{n_{gas}}{E} \frac{d}{dt} \left[ \frac{1}{N_0} \sum_{i=1}^{N_0} r_{  ,0i} \right]$ where $N_0$ is the particle count of the 0 <sup>th</sup> species, $E$ is field magnitude, $n_{gas}$ is the density of the background gas, and $r_{  ,0i}$ is displacement component parallel to the E field vector of the $i^{th}$ 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 <code>step = 0</code> , and with $t = DT$ corresponding to <code>step = 1</code> .
t	(float) The time (s) elapsed in the simulation.

## Methods

<code>get_params()</code>	Return the set of parameters and their default values as a python dictionary.
---------------------------	---

### `thunderboltz.parameters.OutputParameters.get_params`

#### `OutputParameters.get_params()`

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

**thunderboltz.parameters.ParticleParameters****class thunderboltz.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 ( $\text{m}^{-3}$ ).
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).
XVX	(float) All 9 components of the position / velocity correlation tensor $\langle \mathbf{rv} \rangle$ are output under XYX, XZY, XVZ, YVX, YVY, YVZ, ZVX, ZVY, and ZVZ.
XX	(float) All 6 components of the symmetric correlation tensor $\langle \mathbf{rr} \rangle$ are output under XX, YY, ZZ, XY, XZ, and YZ.
step	(int) The number of time steps elapsed in the simulation, with $t = 0$ corresponding to $\text{step} = 0$ .
t	(float) The time (s) elapsed in the simulation.

**Methods**

<code>get_params()</code>	Return the set of parameters and their default values as a python dictionary.
---------------------------	---

**thunderboltz.parameters.ParticleParameters.get\_params****ParticleParameters.get\_params()**

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

### 4.1.3 Cumulative Reaction Counts

If cumulative reaction counts are required, they can be accessed easily for each reaction with the `get_counts()` method. This will return a `DataFrame` where each column corresponds to a collision process and each row corresponds to a time step.

### 4.1.4 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:

```
import thunderboltz as tb

calc = tb.ThunderBoltz(
    # For example, using the Helium model.
    indeck=tb.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.

## API REFERENCE

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

### 5.1 API Reference

<i>ThunderBoltz</i> ([directory, cs, out_file, ...])	ThunderBoltz 0D DSMC simulation wrapper.
--	--

#### 5.1.1 thunderboltz.ThunderBoltz

```
class thunderboltz.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** – ThunderBoltz simulation parameters. Any attributes of *TBParameters* or *WrapParameters* can be passed here.

## Attributes

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

## Methods

<code>add_callback(f)</code>	Add a function to the list of functions that will be called during banner output.
<code>compile_debug()</code>	Prepare all files and compile ThunderBoltz with -g debug flag.
<code>compile_from(src_path[, debug])</code>	Copy TB files from src_path and compile in simulation directory.
<code>compute_fit(x_)</code>	Get the slope and associated error of the line of best fit.
<code>describe_dist([steps, sample_cap])</code>	Generate percentile and count statistics of the electron velocity / energy distribution for various time steps.
<code>get_counts()</code>	Return the cumulative reaction counts for each process and each time step.



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

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

**thunderboltz.ThunderBoltz.compile\_debug**

ThunderBoltz.compile\_debug()

Prepare all files and compile ThunderBoltz with -g debug flag.

**thunderboltz.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.

**thunderboltz.ThunderBoltz.compute\_fit**

ThunderBoltz.compute\_fit(*x\_*)

Get the slope and associated error of the line of best fit. If *x* is a Dataframe, do so for each column.

**thunderboltz.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`

**thunderboltz.ThunderBoltz.get\_counts**

`ThunderBoltz.get_counts()`

Return the cumulative reaction counts for each process and each time step.

**Returns**

A table where each column corresponds to a collision process and each row corresponds to a time step.

**Return type**

`pandas.DataFrame`

**thunderboltz.ThunderBoltz.get\_directory**

`ThunderBoltz.get_directory()`

Return the path of the current simulation.

**thunderboltz.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`

**thunderboltz.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`**thunderboltz.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 will have columns with keywords matching the attributes of *ParticleParameters*.**Return type**`list[pandas.DataFrame]`**thunderboltz.ThunderBoltz.get\_sim\_param****ThunderBoltz.get\_sim\_param(key)**

Return the value of a simulation parameter.

**Raises**`IndexError` – if the parameter is not set.**thunderboltz.ThunderBoltz.get\_slope\_fit****ThunderBoltz.get\_slope\_fit()****thunderboltz.ThunderBoltz.get\_ss\_params****ThunderBoltz.get\_ss\_params(ss\_func=None, fits=False)**

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.

**:param ss\_func (callable[pandas.DataFrame: TBParameters]->:class:pandas.DataFrame):**A function that takes in numerical time series data and returns a new frame with only data that is considered to be at steady state. For example, `ss_func=lambda df: df[df.t > 1e-6]` would select only times in the simulation after one microsecond for steady state calculations, or `ss_func=lambda df: df.iloc[50:, :]` would select the last 50 time steps.**:param**`[TBParameters]->:class:pandas.DataFrame):` A function that takes in numerical time series data and returns a new frame with only data that is considered to be at steady state. For example,

`ss_func=lambda df: df[df.t > 1e-6]` would select only times in the simulation after one microsecond for steady state calculations, or `ss_func=lambda df: df.iloc[50:,:]` would select the last 50 time steps.

#### Parameters

**fits** (*bool*) – Option to use a line of best fit over the steady state window to calculate time dependent parameters (bulk swarm parameters and rate coefficients). rather than averaging derivatives with the forward difference formula. True does so for all rate parameters. Default is False.

#### 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 steady state criteria via `ss_func`.

### thunderboltz.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`

### thunderboltz.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.

**thunderboltz.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`

**thunderboltz.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.

**thunderboltz.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.

---

**thunderboltz.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

**thunderboltz.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



**thunderboltz.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*])

**thunderboltz.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"].

**thunderboltz.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.

**thunderboltz.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`**thunderboltz.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`.

---

**thunderboltz.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`

**thunderboltz.ThunderBoltz.reset****ThunderBoltz.reset()**

Reset output data for a new run.

**thunderboltz.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.**thunderboltz.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*.

**thunderboltz.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).

**thunderboltz.ThunderBoltz.set\_ts\_plot\_params****ThunderBoltz.set\_ts\_plot\_params(*params*)**

Set the default series plotted by *plot\_timeseries()*

**thunderboltz.ThunderBoltz.to\_pickleable****ThunderBoltz.to\_pickleable()**

Return a picklable version of this object.

**thunderboltz.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**

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

**thunderboltz.tb.read****thunderboltz.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*

**thunderboltz.tb.query\_tree**

```
thunderboltz.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  $\mu\text{m}$ .
- **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
<code>pandas.DataFrame</code>	Frames will be concatenated row-wise and one larger DataFrame will be returned.
<code>list[pandas.DataFrame]</code>	A list of frames the same length of the return value will be returned. The frame at index <code>i</code> will contain the concatenated data from each simulation returned by <code>callable(tb)[i]</code> .
<code>list[Any]</code>	A list of lists will be returned. The list at index <code>i</code> will contain a list of items returned from each call to <code>callable(tb)[i]</code> .
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]]`

**thunderboltz.tb.CrossSections.from\_LXCat****CrossSections.from\_LXCat**(*fname*)

Load cross section data from an LXCat .txt file

**5.1.3 Cross Sections**

<i>CrossSections</i> ([directory, input_path, ...])	ThunderBoltz cross section set data type.
<i>Process</i> (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.
<i>input.He_TB</i> ([n, egen, analytic_cs, eadf, ...])	Generate parameterized He cross section sets in the ThunderBoltz format.
<i>input.convert</i> (df, u1, u2[, inv, drop, add])	Convert easily between units with a labeled DataFrame.

**thunderboltz.CrossSections**

**class** thunderboltz.**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

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

## Methods

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

### thunderboltz.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.

**thunderboltz.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.**thunderboltz.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.**thunderboltz.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.** –

**thunderboltz.CrossSections.from\_LXCat****CrossSections.from\_LXCat(*fname*)**

Load cross section data from an LXCat .txt file

**thunderboltz.CrossSections.get\_deck****CrossSections.get\_deck()**

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

**thunderboltz.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  $\epsilon$  on the x-axis.
- **\*\*plot\_args** – Optional arguments passed to Axes.plot().

**Returns**

The axes object.

**Return type**

ax (`matplotlib.axes.Axes`)

**thunderboltz.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.

**thunderboltz.CrossSections.set\_fixed\_background**

`CrossSections.set_fixed_background(fixed=True)`

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

**thunderboltz.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.

**thunderboltz.Process**

**class** thunderboltz.**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  $\text{m}^2$  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 ( $\text{m}^2$ ).

**Attributes**

SAMPLE_MAX
SAMPLE_MIN
cs_func
nsamples

**Methods**

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

**thunderboltz.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.

**thunderboltz.Process.auto\_sample****Process.auto\_sample()**

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

**thunderboltz.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.**thunderboltz.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.

**thunderboltz.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)***thunderboltz.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

**thunderboltz.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.

**thunderboltz.input.He\_TB**

`thunderboltz.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]

**thunderboltz.input.convert**

`thunderboltz.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

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

### thunderboltz.parallel.MPRunner

**class** thunderboltz.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

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

### thunderboltz.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.

### thunderboltz.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.

**thunderboltz.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.**thunderboltz.parallel.MPRunner.to\_pickleable****MPRunner.to\_pickleable**()

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

**thunderboltz.parallel.DistributedPool****class** thunderboltz.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**

<a href="#">err_callback</a> (err)	Print out errors that subprocesses encounter.
<a href="#">submit</a> ([run_args])	Submit a single job with updated key words to the pool.

**thunderboltz.parallel.DistributedPool.err\_callback****DistributedPool.err\_callback**(err)

Print out errors that subprocesses encounter.

**thunderboltz.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\(\)](#).

**thunderboltz.parallel.SlurmManager**

**class** thunderboltz.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 <i>MPRunner</i> object.
job_ids	Store references to the slurm job numbers after jobs are submitted
options	The SLURM sbatch options

**Methods**

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

### `thunderboltz.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.

### `thunderboltz.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)

### `thunderboltz.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)

### `thunderboltz.parallel.SlurmManager.join`

`SlurmManager.join()`

Wait for all slurm jobs to finish.

### `thunderboltz.parallel.SlurmManager.mock_run`

`SlurmManager.mock_run()`

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



**thunderboltz.parallel.SlurmManager.process\_batch\_script**

`SlurmManager.process_batch_script()`

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

**thunderboltz.parallel.SlurmManager.sbatch**

`SlurmManager.sbatch()`

Call slurm with current settings.

**thunderboltz.parallel.SlurmManager.set\_**

`SlurmManager.set_(*options)`

Update slurm manager options.

**Parameters**

**\*\*options** – SLURM settings.

**thunderboltz.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()`.

**thunderboltz.parallel.SlurmManager.write\_slurm\_script**

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

Write the SLURM batch script.

## A

add\_callback() (*thunderboltz.ThunderBoltz* method), 33  
 add\_differential\_model() (*thunderboltz.CrossSections* method), 46  
 add\_differential\_parameters() (*thunderboltz.Process* method), 49  
 add\_process() (*thunderboltz.CrossSections* method), 47  
 add\_processes() (*thunderboltz.CrossSections* method), 47  
 auto\_sample() (*thunderboltz.Process* method), 50

## B

batch\_script() (*thunderboltz.parallel.SlurmManager* method), 55

## C

compile\_debug() (*thunderboltz.ThunderBoltz* method), 33  
 compile\_from() (*thunderboltz.ThunderBoltz* method), 33  
 compute\_fit() (*thunderboltz.ThunderBoltz* method), 33  
 convert() (in module *thunderboltz.input*), 51  
 CrossSections (class in *thunderboltz*), 45

## D

describe\_dist() (*thunderboltz.ThunderBoltz* method), 33  
 DistributedPool (class in *thunderboltz.parallel*), 53

## E

err\_callback() (*thunderboltz.parallel.DistributedPool* method), 53

## F

find\_infile() (*thunderboltz.CrossSections* method), 47  
 from\_LXCat() (*thunderboltz.CrossSections* method), 47  
 from\_LXCat() (*thunderboltz.tb.CrossSections* method), 45

## G

get\_counts() (*thunderboltz.ThunderBoltz* method), 34  
 get\_deck() (*thunderboltz.CrossSections* method), 47  
 get\_directory() (*thunderboltz.parallel.MPRunner* method), 52  
 get\_directory() (*thunderboltz.ThunderBoltz* method), 34  
 get\_edfs() (*thunderboltz.ThunderBoltz* method), 34  
 get\_etrans() (*thunderboltz.ThunderBoltz* method), 35  
 get\_params() (*thunderboltz.parameters.OutputParameters* method), 27  
 get\_params() (*thunderboltz.parameters.ParticleParameters* method), 28  
 get\_params() (*thunderboltz.parameters.TBParameters* method), 22  
 get\_params() (*thunderboltz.parameters.WrapParameters* method), 24  
 get\_particle\_tables() (*thunderboltz.ThunderBoltz* method), 35  
 get\_sim\_param() (*thunderboltz.ThunderBoltz* method), 35  
 get\_slope\_fit() (*thunderboltz.ThunderBoltz* method), 35  
 get\_ss\_params() (*thunderboltz.ThunderBoltz* method), 35  
 get\_timeseries() (*thunderboltz.ThunderBoltz* method), 36  
 get\_vdfs() (*thunderboltz.ThunderBoltz* method), 36

## H

has\_active() (*thunderboltz.parallel.SlurmManager* method), 55  
 has\_pending() (*thunderboltz.parallel.SlurmManager* method), 55  
 He\_TB() (in module *thunderboltz.input*), 51

## J

join() (*thunderboltz.parallel.SlurmManager* method), 55

## M

`mock_run()` (*thunderboltz.parallel.SlurmManager method*), 55  
`MPRunner` (class in *thunderboltz.parallel*), 52

## O

`OutputParameters` (class in *thunderboltz.parameters*), 24

## P

`ParticleParameters` (class in *thunderboltz.parameters*), 27  
`plot_cs()` (*thunderboltz.CrossSections method*), 47  
`plot_cs()` (*thunderboltz.ThunderBoltz method*), 37  
`plot_edf_comps()` (*thunderboltz.ThunderBoltz method*), 38  
`plot_edfs()` (*thunderboltz.ThunderBoltz method*), 38  
`plot_rates()` (*thunderboltz.ThunderBoltz method*), 39  
`plot_timeseries()` (*thunderboltz.ThunderBoltz method*), 39  
`plot_vdfs()` (*thunderboltz.ThunderBoltz method*), 40  
`Process` (class in *thunderboltz*), 48  
`process_batch_script()` (*thunderboltz.parallel.SlurmManager method*), 56

## Q

`query_tree()` (in module *thunderboltz.tb*), 44

## R

`read()` (in module *thunderboltz.tb*), 43  
`read()` (*thunderboltz.CrossSections method*), 48  
`read()` (*thunderboltz.ThunderBoltz method*), 40  
`read_log()` (*thunderboltz.ThunderBoltz method*), 40  
`read_particle_table()` (*thunderboltz.ThunderBoltz method*), 41  
`read_stdout()` (*thunderboltz.ThunderBoltz method*), 41  
`read_tb_params()` (*thunderboltz.ThunderBoltz method*), 41  
`require_cs()` (*thunderboltz.Process method*), 50  
`reset()` (*thunderboltz.ThunderBoltz method*), 42  
`run()` (*thunderboltz.parallel.MPRunner method*), 52  
`run()` (*thunderboltz.ThunderBoltz method*), 42

## S

`sample_cs()` (*thunderboltz.Process method*), 50  
`sbatch()` (*thunderboltz.parallel.SlurmManager method*), 56  
`set_()` (*thunderboltz.parallel.MPRunner method*), 53  
`set_()` (*thunderboltz.parallel.SlurmManager method*), 56  
`set_()` (*thunderboltz.ThunderBoltz method*), 42  
`set_fixed_background()` (*thunderboltz.CrossSections method*), 48

`set_fixed_tracking()` (*thunderboltz.ThunderBoltz method*), 43  
`set_ts_plot_params()` (*thunderboltz.ThunderBoltz method*), 43  
`SlurmManager` (class in *thunderboltz.parallel*), 54  
`submit()` (*thunderboltz.parallel.DistributedPool method*), 53  
`submit()` (*thunderboltz.parallel.SlurmManager method*), 56

## T

`TBParameters` (class in *thunderboltz.parameters*), 20  
`ThunderBoltz` (class in *thunderboltz*), 30  
`to_cs_frame()` (*thunderboltz.Process method*), 50  
`to_df()` (*thunderboltz.Process method*), 50  
`to_pickleable()` (*thunderboltz.parallel.MPRunner method*), 53  
`to_pickleable()` (*thunderboltz.ThunderBoltz method*), 43

## W

`WrapParameters` (class in *thunderboltz.parameters*), 22  
`write()` (*thunderboltz.CrossSections method*), 48  
`write_input()` (*thunderboltz.ThunderBoltz method*), 43  
`write_slurm_script()` (*thunderboltz.parallel.SlurmManager method*), 56

## Z

`zero_below_thresh()` (*thunderboltz.Process method*), 51