turboPy

Release v2023.06.09

Steve Richardson

Jun 09, 2023

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 turboPy Conda environment

- Create a conda environment for turboPy: `conda env create -f environment.yml`
- Activate: `conda activate turbopy`
- **Install turboPy into the environment (from the main folder where `setup.py` is):**
  - Install turboPy in editable mode (i.e. setuptools "develop mode") if you are modifying turboPy itself: `pip install -e .`
  - If you just plan to develop a code using the existing turboPy framework: `pip install .`
- Run tests: `pytest`

1.2 turboPy development environment

If using `pylint` (which you should!) add `variable-rgx=[a-z0-9_]{1,30}$` to your `.pylintrc` file to allow single character variable names.

Merge requests are encouraged!

1.3 Example turboPy app

Once you have the turboPy conda environment set up, you can go ahead and write a "turboPy app". The simplest way to get started with writing an app might be to clone an existing example app.

This example app computes the motion of a charged particle in an electric field.

CHAPTER

TWO

SHARING RESOURCES

It is often necessary to share resources between custom `turbopy.core.PhyiscsModules` or `turbopy.core.Diagnostics`. A new API has been developed to assist with this. To use this new API, you simply need to define a couple of dictionaries (`_resources_to_share` and `_needed_resources`) in your class. Then, when the `prepare_simulation` method of your simulation is called, the shared variables get set up automatically.

2.1 Making resources available to other modules

In order to tell other `turbopy.core.PhyiscsModules` about resources that you want to share, just add them to the member variable `_resources_to_share` in the `__init__` method. For example, the following function will share the variables `self.position` and `self.momentum`:

```
def __init__(self, owner: Simulation, input_data: dict):
    super().__init__(owner, input_data)
    self.position = np.zeros((1, 3))
    self.momentum = np.zeros((1, 3))

    self._resources_to_share = {"position": self.position,
                                "momentum": self.momentum}
```

Note that the variables that you want to share need to be defined before they can be added to the `_resources_to_share` dictionary. Also, make sure that they are *mutable* variables, otherwise other modules won't see any changes that you make to them during the simulation.

2.2 Looking for shared resources

If your module needs access to a variable that is being shared from a different module, you use the member variable `_needed_resources`. In this example, the data shared by the example above will be saved.

```
def __init__(self, owner: Simulation, input_data: dict):
    super().__init__(owner, input_data)
    self._needed_resources = {"position": "x",
                              "momentum": "p"}
```

This will create the variables `self.x` and `self.p`, which will point to the position and momentum data shared by the second module.

CHAPTER

THREE

TURBOPY API

The core turboPy API is composed of one main class (the *turbopy.core.Simulation* class) and three abstract base classes, `turbopy.core.PhyiscsModule`, *turbopy.core.Diagnostic*, and *turbopy.core.ComputeTool*.

3.1 Core framework classes

Core base classes of the turboPy framework

Notes

The published paper for Turbopy: A lightweight python framework for computational physics can be found in the link below[1].

References

class turbopy.core.**ComputeTool**(*owner:* Simulation, *input_data: dict*)

Bases: *DynamicFactory*

This is the base class for compute tools

These are the compute-heavy functions, which have implementations of numerical methods which can be shared between physics modules.

Parameters

- **owner** (*Simulation*) – Simulation class that ComputeTool belongs to.
- **input_data** (*dict*) – Dictionary that contains user defined parameters about this object such as its name.

_registry

Registered derived ComputeTool classes.

Type
dict

_factory_type_name

Type of ComputeTool child class

Type
str

[1] 1 A.S. Richardson, D.F. Gordon, S.B. Swanekamp, I.M. Rittersdorf, P.E. Adamson, O.S. Grannis, G.T. Morgan, A. Ostenfeld, K.L. Phlips, C.G. Sun, G. Tang, and D.J. Watkins, Comput. Phys. Commun. 258, 107607 (2021). https://doi.org/10.1016/j.cpc.2020.107607

_owner

Simulation class that ComputeTool belongs to.

Type
Simulation

_input_data

Dictionary that contains user defined parameters about this object such as its name.

Type
dict

name

Type of ComputeTool.

Type
str

custom_name

Name given to individual instance of tool, optional. Used when multiple tools of the same type exist in one *Simulation*.

Type
str

initialize()

Perform any initialization operations needed for this tool

class turbopy.core.**Diagnostic**(*owner:* Simulation, *input_data: dict*)

Bases: *DynamicFactory*

Base diagnostic class.

Parameters

- **owner** (Simulation) – The Simulation object that owns this object
- **input_data** (*dict*) – Dictionary that contains user defined parameters about this object such as its name.

_factory_type_name

Type of DynamicFactory child class

Type
str

_registry

Registered derived Diagnostic classes

Type
dict

_owner

The Simulation object that contains this object

Type
Simulation

_input_data

Dictionary that contains user defined parameters about this object such as its name.

Type
dict

`_needed_resources`

Dictionary that lists shared resources that this module needs. Format is *{shared_key: variable_name}*, where *shared_key* is a string with the name of needed resource, and *variable_name* is a string to use when saving this variable. For example: {"Fields:E": "E"} will make *self.E*.

Type
dict

`diagnose()`

Perform diagnostic step

This gets called on every step of the main simulation loop.

Raises
`NotImplementedError` – Method or function hasn't been implemented yet. This is an abstract base class. Derived classes must implement this method in order to be a concrete child class of *Diagnostic*.

`finalize()`

Perform any finalization operations

This gets called once after the main simulation loop is complete.

`initialize()`

Perform any initialization operations

This gets called once before the main simulation loop. Base class definition creates output directory if it does not already exist. If subclass overrides this function, call *super().initialize()*

`inspect_resource(resource: dict)`

Deprecated

This method is only here for backwards compatability. New code should use the ``_needed_resources`` dictionary.

Save references to data from other PhysicsModules If your subclass needs the data described by the key, now's their chance to save a reference to the data :param resource: A dictionary containing references to data shared by other

PhysicsModules.

`class turbopy.core.DynamicFactory`

Bases: `ABC`

Abstract class which provides dynamic factory functionality

This base class provides a dynamic factory pattern functionality to classes that derive from this.

`classmethod is_valid_name(name: str)`

Check if the name is in the registry

`classmethod lookup(name: str)`

Look up a name in the registry, and return the associated derived class

classmethod register(*name_to_register: str*, *class_to_register*, *override=False*)

Add a derived class to the registry

class turbopy.core.**Grid**(*input_data: dict*)

Bases: object

Grid class

Parameters

input_data (*dict*) – Dictionary containing parameters needed to defined the grid. Currently only 1D grids are defined in turboPy.

The expected parameters are:

- **"N" | {"dr" | "dx"} :**
  The number of grid points (*int*) | the grid spacing (*float*)
- **"min" | "x_min" | "r_min" :**
  The coordinate value of the minimum grid point (*float*)
- **"max" | "x_max" | "r_max" :**
  The coordinate value of the maximum grid point (*float*)

_input_data

Dictionary containing parameters needed to defined the grid. Currently only 1D grids are defined in turboPy.

Type
dict

r_min

Min of the Grid range.

Type
float, None

r_max

Max of the Grid range.

Type
float, None

num_points

Number of points on Grid.

Type
int, None

dr

Grid spacing.

Type
float, None

r, cell_edges

Array of evenly spaced Grid values.

Type
numpy.ndarray

cell_centers

Value of the coordinate in the middle of each Grid cell.

Type
float

cell_widths

Width of each cell in the Grid.

Type
numpy.ndarray

r_inv

Inverse of coordinate values at each Grid point, 1/`Grid.r`.

Type
float

create_interpolator(*r0*)

Return a function which linearly interpolates any field on this grid, to the point `r0`.

Parameters
r0 (*float*) – The requested point on the grid.

Returns
A function which takes a grid quantity `y` and returns the interpolated value of `y` at the point `r0`.

Return type
function

generate_field(*num_components=1*, *placement_of_points='edge-centered'*)

Returns squeezed numpy.ndarray of zeros with dimensions *Grid.num_points* and *num_components*.

Parameters

- **num_components** (*int, defaults to 1*) – Number of vector components at each point.
- **placement_of_points** (*str, defaults to "edge-centered"*) – Designate position of points on grid

Returns
Squeezed array of zeros.

Return type
numpy.ndarray

generate_linear()

Returns numpy.ndarray with *Grid.num_points* evenly spaced in the interval between 0 and 1.

numpy.ndarray
Evenly spaced array.

parse_grid_data()

Initializes the grid spacing, range, and number of points on the grid from *Grid._input_data*.

Raises
RuntimeError – If the range and step size causes a non-integer number of grid points.

set_value_from_keys(*var_name*, *options*)

Initializes a specified attribute to a value provided in *Grid._input_data*.

Parameters

- **var_name** (*str*) – Attribute name to be initialized.
- **options** (*set*) – Set of keys in *Grid._input_data* to search for values.

Raises
KeyError – If none of the keys in *options* are present in *Grid._input_data*.

class turbopy.core.**PhysicsModule**(*owner: Simulation*, *input_data: dict*)

Bases: *DynamicFactory*

This is the base class for all physics modules

By default, a subclass will share any public attributes as turboPy resources. The default resource name for these automatically shared attributes is the string form by combining the class name and the attribute name: *<class_name>_<attribute_name>*.

If there are attributes that should not be automatically shared, then use the python "private" naming convention, and give the attribute a name which starts with an underscore.

Parameters

- **owner** (*Simulation*) – Simulation class that *PhysicsModule* belongs to.
- **input_data** (*dict*) – Dictionary that contains user defined parameters about this object such as its name.

_owner

Simulation class that PhysicsModule belongs to.

Type
Simulation

_module_type

Module type.

Type
str, None

_input_data

Dictionary that contains user defined parameters about this object such as its name.

Type
dict

_registry

Registered derived ComputeTool classes.

Type
dict

_factory_type_name

Type of PhysicsModule child class.

Type
str

_needed_resources

Dictionary that lists shared resources that this module needs. Format is *{shared_key: variable_name}*, where *shared_key* is a string with the name of needed resource, and *variable_name* is a string to use when saving this variable. For example: {"Fields:E": "E"} will make *self.E*.

Type
dict

_resources_to_share

Dictionary that lists shared resources that this module is sharing to others. Format is *{shared_key: variable}*, where *shared_key* is a string with the name of resource to share, and *variable* is the data to be shared.

Type
dict

Notes

This class is based on Module class in TurboWAVE. Because python mutable/immutable is different than C++ pointers, the implementation here is different. Here, a "resource" is a dictionary, and can have more than one thing being shared. Note that the value stored in the dictionary needs to be mutable. Make sure not to reinitialize it, because other physics modules will be holding a reference to it.

exchange_resources()

Main method for sharing resources with other *PhysicsModule* objects.

This is the function where you call *publish_resource()*, to tell other physics modules about data you want to share.

By default, any "public" attributes (those with names that do not start with an underscore) will be shared with the key *<class_name>_<attribute_name>*.

initialize()

Perform initialization operations for this *PhysicsModule*

This is called before the main simulation loop

inspect_resource(*resource: dict*)

Deprecated

This method is only here for backwards compatability. New code should use the ``_needed_resources`` dictionary.

Method for accepting resources shared by other PhysicsModules If your subclass needs the data described by the key, now's their chance to save a pointer to the data. :param resource: resource dictionary to be shared :type resource: *dict*

publish_resource(*resource: dict*)

Deprecated

This method is only here for backwards compatability. New code should use the ``_resources_to_share`` dictionary.

Method which implements the details of sharing resources :param resource: resource dictionary to be shared :type resource: *dict*

reset()

Perform any needed reset operations

This is called at every time step in the main loop, before any of the calls to *update*.

update()

Do the main work of the *PhysicsModule*

This is called at every time step in the main loop.

class turbopy.core.**Simulation**(*input_data: dict*)

Bases: object

Main turboPy simulation class

This Class "owns" all the physics modules, compute tools, and diagnostics. It also coordinates them. The main simulation loop is driven by an instance of this class.

Parameters

input_data (*dict*) – This dictionary contains all parameters needed to set up a turboPy simulation. Each key describes a section, and the value is another dictionary with the needed parameters for that section.

Expected keys are:

"Grid", optional
Dictionary containing parameters needed to define the grid. Currently only 1D grids are defined in turboPy.

The expected parameters are:

- **"N" | {"dr" | "dx"} :**
  The number of grid points (*int*) | the grid spacing (*float*)
- **"min" | "x_min" | "r_min" :**
  The coordinate value of the minimum grid point (*float*)
- **"max" | "x_max" | "r_max" :**
  The coordinate value of the maximum grid point (*float*)

"Clock"
Dictionary of parameters needed to define the simulation clock.

The expected parameters are:

- **"start_time" :**
  The time for the start of the simulation (*float*)
- **"end_time" :**
  The time for the end of the simulation (*float*)
- **"num_steps" | "dt" :**
  The number of time steps (*int*) | the size of the time step (*float*)
- **"print_time" :**
  bool, optional, default is `False`

"PhysicsModules"
[*dict* [*str*, *dict*]] Dictionary of *PhysicsModule* items needed for the simulation.

Each key in the dictionary should map to a *PhysicsModule* subclass key in the *PhysicsModule* registry.

The value is a dictionary of parameters which is passed to the constructor for the *PhysicsModule*.

"Diagnostics"
[*dict* [*str*, *dict*], optional] Dictionary of *Diagnostic* items needed for the simulation.

Each key in the dictionary should map to a *Diagnostic* subclass key in the *Diagnostic* registry.

The value is a dictionary of parameters which is passed to the constructor for the *Diagnostic*.

If the key is not found in the registry, then the key/value pair is interpreted as a default parameter value, and is added to dictionary of parameters for all of the *Diagnostic* constructors.

If the directory and filename keys are not specified, default values are created in the *read_diagnostics_from_input()* method. The default name for the directory is "default_output" and the default filename is the name of the Diagnostic subclass followed by a number.

"Tools"
[*dict* [*str*, *dict*], optional] Dictionary of *ComputeTool* items needed for the simulation.

Each key in the dictionary should map to a *ComputeTool* subclass key in the *ComputeTool* registry.

The value is a dictionary of parameters which is passed to the constructor for the *ComputeTool*.

physics_modules
A list of *PhysicsModule* objects for this simulation.

Type
list of *PhysicsModule* subclass objects

diagnostics
A list of *Diagnostic* objects for this simulation.

Type
list of *Diagnostic* subclass objects

compute_tools
A list of *ComputeTool* objects for this simulation.

Type
list of *ComputeTool* subclass objects

finalize_simulation()
Close out the simulation

Runs the *Diagnostic.finalize()* method for each diagnostic.

find_tool_by_name(*tool_name: str*, *custom_name: Optional[str] = None*)
Returns the *ComputeTool* associated with the given name

fundamental_cycle()
Perform one step of the main time loop

Executes each diagnostic and physics module, and advances the clock.

prepare_simulation()
Prepares the simulation by reading the input and initializing physics modules and diagnostics.

read_clock_from_input()
Construct the clock based on input parameters

read_diagnostics_from_input()
Construct *Diagnostic* instances based on input

read_grid_from_input()
Construct the grid based on input parameters

read_modules_from_input()

Construct *PhysicsModule* instances based on input

read_tools_from_input()

Construct `ComputeTools` based on input

run()

Runs the simulation

This initializes the simulation, runs the main loop, and then finalizes the simulation.

sort_modules()

Sort *Simulation.physics_modules* by some logic

Unused stub for future implementation

class turbopy.core.**SimulationClock**(*owner:* Simulation, *input_data: dict*)

Bases: `object`

Clock class for turboPy

Parameters

- **owner** (*Simulation*) – Simulation class that SimulationClock belongs to.
- **input_data** (*dict*) – Dictionary of parameters needed to define the simulation clock.

  The expected parameters are:

  - **"start_time" :**
    The time for the start of the simulation (*float*)
  - **"end_time" :**
    The time for the end of the simulation (*float*)
  - **"num_steps" | "dt" :**
    The number of time steps (*int*) | the size of the time step (*float*)
  - **"print_time" :**
    bool, optional, default is `False`

_owner

Simulation class that SimulationClock belongs to.

Type
Simulation

_input_data

Dictionary of parameters needed to define the simulation clock.

Type
dict

start_time

Clock start time.

Type
float

time

Current time on clock.

Type
float

end_time

Clock end time.

Type
float

this_step

Current time step since start.

Type
int

print_time

If True will print current time after each increment.

Type
bool

num_steps

Number of steps clock will take in the interval.

Type
int

dt

Time passed at each increment.

Type
float

advance()

Increment the time

is_running()

Check if time is less than end time

turn_back(*num_steps=1*)

Set the time back *num_steps* time steps

3.2 Diagnostic classes

Diagnostics module for the turboPy computational physics simulation framework.

Diagnostics can access `PhysicsModule` data. They are called every time step, or every N steps. They can write to file, cache for later, update plots, etc, and they can halt the simulation if conditions require.

class turbopy.diagnostics.**CSVOutputUtility**(*filename*, *diagnostic_size*, ***kwargs*)

Bases: *OutputUtility*

Comma separated value (CSV) diagnostic output helper class

Provides routines for writing data to a file in CSV format. This class can be used by Diagnostics subclassses to handle output to csv format.

Parameters

- **filename** (*str*) – File name for CSV data file.

- **diagnostic_size** ((*int*, *int*)) – Size of data set to be written to CSV file. First value is the number of time points. Second value is number of spatial points.

filename

File name for CSV data file.

Type
str

buffer

Buffer for storing data before it is written to file.

Type
numpy.ndarray

buffer_index

Position in buffer.

Type
int

append(*data*)

Append data to the buffer.

Deprecated since version `append`: has been removed from the public API. Use *diagnose* instead.

diagnose(*data*)

Adds 'data' into csv output buffer.

Parameters
data (numpy.ndarray) – 1D numpy array of values to be added to the buffer.

finalize()

Write the CSV data to file.

write_data()

Write buffer to file

class turbopy.diagnostics.**ClockDiagnostic**(*owner: Simulation*, *input_data: dict*)

Bases: *Diagnostic*

Diagnostic subclass used to store and save time data into a CSV file using the CSVOutputUtility class.

Parameters

- **owner** (Simulation) – The Simulation object that contains this object
- **input_data** (*dict*) – Dictionary containing information about this diagnostic such as its name

owner

The Simulation object that contains this object

Type
Simulation

input_data

Dictionary containing information about this diagnostic such as its name

Type
dict

filename

File name for CSV time file

Type
str

csv

Array to store values to be written into a CSV file

Type
numpy.ndarray

interval

The time interval to wait in between writing to output file. If interval is None, then the outputs are written only at the end of the simulation.

Type
float, None

handler

The *IntervalHandler* object that handles writing to output files while the simulation is running. Is None if the interval parameter is not specified

Type
IntervalHandler

diagnose()

Append time into the csv buffer.

finalize()

Write time into self.csv and saves as a CSV file.

initialize()

Initialize *self.csv* as an instance of the CSVOuputUtility class.

class turbopy.diagnostics.**FieldDiagnostic**(*owner:* Simulation, *input_data: dict*)

Bases: *Diagnostic*

Parameters

- **owner** (Simulation) – Simulation object containing current object.
- **input_data** (*dict*) – Dictionary that contains information regarding location, field, and output type.

component

Type
str

field_name

Field.

Type
str

output

Output type.

Type
str

field

Field as dictated by resource.

Type
str, None

dump_interval

Time interval at which the diagnostic is run.

Type
int, None

write_interval

Time interval at which the diagnostic buffer is written to file. If this is None, then the buffer is not written out until the end of the simulation.

Type
int, None

diagnose

Uses the dump and write handlers to perform the diagnostic actions.

Type
method

diagnostic_size

Size of data set to be written to CSV file. First value is the number of time points. Second value is number of spatial points.

Type
(int, int), None

diagnose()

Perform diagnostic step

This gets called on every step of the main simulation loop.

Raises
NotImplementedError – Method or function hasn't been implemented yet. This is an abstract base class. Derived classes must implement this method in order to be a concrete child class of `Diagnostic`.

do_diagnostic()

Run output_function depending on field.shape.

finalize()

Write the CSV data to file if CSV is the proper output type.

initialize()

Initialize diagnostic_size and output function if provided as csv, and self.csv as an instance of the *CSVOutputUtility* class.

class turbopy.diagnostics.**GridDiagnostic**(*owner:* Simulation, *input_data: dict*)

Bases: *Diagnostic*

Diagnostic subclass used to store and save grid data into a CSV file

Parameters

- **owner** (Simulation) – The 'Simulation' object that contains this object

- **input_data** (*dict*) – Dictionary containing information about this diagnostic such as its name

owner

The 'Simulation' object that contains this object

Type
Simulation

input_data

Dictionary containing information about this diagnostic such as its name

Type
dict

filename

File name for CSV grid file

Type
str

diagnose()

Grid diagnotic only runs at startup

initialize()

Save grid data into CSV file

class turbopy.diagnostics.**HistoryDiagnostic**(*owner:* Simulation, *input_data: dict*)

Bases: *Diagnostic*

Outputs histories/traces as functions of time

This diagnostic assists in outputting 1D history traces. Multiple time- dependant quantities can be selected, and are output to a NetCDF file using the xarray python package.

Examples

When using a python dictionary to define the turboPy simulation, the history diagnostics can be added as in this example. Each item in the "traces" list has several key: value pairs. The "name" key corresponds to a turboPy resource that is shared by another module. The "coords" key is used in cases where the shared resource is more than just a scalar quantitiy. In this example, the position and momentum are length-3 vectors, with the three entries corresponding to the three vector components. In the case where a resources is a quantity on the grid, then something like `'coords': ['x'], 'units': 'm'` might be appropriate.

Note that the 'coords' list has two items, because the shape of the shared numpy array is `(1, 3)` in this example. The first item is basically just a placeholder, and is called "dim0".

```
>>> simulation_parameters = {"Diagnostics": {
            "histories": {
                "filename": "output.nc",
                "traces": [
                    {'name': 'EMField:E'},
                    {'name': 'ChargedParticle:momentum',
                    'units': 'kg m/s',
                    'coords': ["dim0", "vector component"],
                    'long_name': 'Particle Momentum'
                    },
```

(continues on next page)

(continued from previous page)

```
                    {'name': 'ChargedParticle:position',
                    'units': 'm',
                    'coords': ["dim0", "vector component"],
                    'long_name': 'Particle Position'
                    },
                ]
            }
        }
    }
```

This is another example of a similar history setup, but in the format expected for a `toml` input file.

```
[Diagnostics.histories]
filename = "history.nc"

[[Diagnostics.histories.traces]]
name = 'ChargedParticle:momentum'
units = 'kg m/s'
coords = ["dim0", "vector component"]
long_name = 'Particle Momentum'

[[Diagnostics.histories.traces]]
name = 'ChargedParticle:position'
units = 'm'
coords = ["dim0", "vector component"]
long_name = 'Particle Position'

[[Diagnostics.histories.traces]]
name = 'EMField:E'
```

References

[1] C. Birdsall and A. Langdon. Plasma Physics via Computer Simulation. Institute of Physics Series in Plasma Physics and Fluid Dynamics. Taylor & Francis, 2004. Page 382.

diagnose()

Perform diagnostic step

This gets called on every step of the main simulation loop.

Raises

NotImplementedError – Method or function hasn't been implemented yet. This is an abstract base class. Derived classes must implement this method in order to be a concrete child class of `Diagnostic`.

finalize()

Perform any finalization operations

This gets called once after the main simulation loop is complete.

initialize()

Perform any initialization operations

This gets called once before the main simulation loop. Base class definition creates output directory if it does not already exist. If subclass overrides this function, call *super().initialize()*

class turbopy.diagnostics.**IntervalHandler**(*interval*, *action*)

Bases: object

Calls a function (action) if a given interval has passed

Parameters

- **interval** (*float, None*) – The time interval to wait in between actions. If interval is None, then the action will be called every time.
- **action** (*callable*) – The function to call when the interval has passed

perform_action(*time*)

Perform the action if an interval has passed

class turbopy.diagnostics.**NPYOutputUtility**(*filename*, *diagnostic_size*, ***kwargs*)

Bases: *OutputUtility*

NumPy formatted binary file (.npy) diagnostic output helper class

Provides routines for writing data to a file in NumPy format. This class can be used by Diagnostics subclassses to handle output to .npy format.

Parameters

- **filename** (*str*) – File name for .npy data file.
- **diagnostic_size** (*(int, int)*) – Size of data set to be written to .npy file. First value is the number of time points. Second value is number of spatial points.

filename

File name for .npy data file.

Type
str

buffer

Buffer for storing data before it is written to file.

Type
numpy.ndarray

buffer_index

Position in buffer.

Type
int

diagnose(*data*)

Adds 'data' into npy output buffer.

Parameters
data (numpy.ndarray) – 1D numpy array of values to be added to the buffer.

finalize()

Write the npy data to file.

write_data()

Write buffer to file

class turbopy.diagnostics.**OutputUtility**(*input_data*)

Bases: ABC

Abstract base class for output utility

An instance of an OutputUtility can (optionally) be used by diagnostic classes to assist with the implementation details needed for outputing the diagnostic information.

abstract diagnose(*data*)

Perform the diagnostic

abstract finalize()

Perform any finalization steps when the simulation is complete

abstract write_data()

Optional function for writting buffer to file etc.

class turbopy.diagnostics.**PointDiagnostic**(*owner: Simulation*, *input_data: dict*)

Bases: *Diagnostic*

Parameters

- **owner** (Simulation) – Simulation object containing current object.
- **input_data** (*dict*) – Dictionary that contains information regarding location, field, and output type.

location

Location.

Type
str

field_name

Field name.

Type
str

output

Output type.

Type
str

get_value

Function to get value given the field.

Type
function, None

field

Field as dictated by resource.

Type
str, None

output_function

Function for assigned output method: standard output or csv.

Type
function, None

csv

numpy.ndarray being written as a csv file.

Type
numpy.ndarray, None

diagnose()

Run output function given the value of the field.

finalize()

Write the CSV data to file if CSV is the proper output type.

initialize()

Initialize output function if provided as csv, and self.csv as an instance of the CSVOuputUtility class.

class turbopy.diagnostics.**PrintOutputUtility**(*input_data*)

Bases: *OutputUtility*

OutputUtility which writes to the screen

diagnose(*data*)

Prints out data to standard output.

Parameters
data (numpy.ndarray) – 1D numpy array of values.

3.3 Compute tools

Several subclasses of the *turbopy.core.ComputeTool* class for common scenarios

Included stock subclasses:

- Solver for the 1D radial Poisson's equation
- Helper functions for constructing sparse finite difference matrices
- Charged particle pusher using the Boris method
- Interpolate a function y(x) given y on a grid in x

class turbopy.computetools.**BorisPush**(*owner:* Simulation, *input_data: dict*)

Bases: *ComputeTool*

Calculate charged particle motion in electric and magnetic fields

This is an implementation of the Boris push algorithm.

Parameters

- **owner** (Simulation) – The *turbopy.core.Simulation* object that contains this object
- **input_data** (*dict*) – There are no custom configuration options for this tool

c2

The speed of light squared

Type
float

push(*position*, *momentum*, *charge*, *mass*, *E*, *B*)

Update the position and momentum of a charged particle in an electromagnetic field

Parameters

- **position** (numpy.ndarray) – The initial position of the particle as a vector
- **momentum** (numpy.ndarray) – The initial momentum of the particle as a vector
- **charge** (*float*) – The electric charge of the particle
- **mass** (*float*) – The mass of the particle
- **E** (numpy.ndarray) – The value of the electric field at the particle
- **B** (numpy.ndarray) – The value of the magnetic field at the particle

class turbopy.computetools.**FiniteDifference**(*owner:* Simulation, *input_data: dict*)

Bases: *ComputeTool*

Helper functions for constructing finite difference matrices

This class contains functions for constructing finite difference approximations to various differential operators. The scipy.sparse package from scipy is used since most of these are tridiagonal sparse matrices.

Parameters

- **owner** (Simulation) – The *turbopy.core.Simulation* object that contains this object
- **input_data** (*dict*) – Dictionary of configuration options. The expected parameters are:
  - **"method" | {"centered" | "upwind_left"} :**
    Select between centered difference, and left upwind difference for the *setup_ddx* member function.

BC_left_avg()

Sparse matrix to set average solution at left boundary

Returns
Matrix which implements a boundary condition for the left boundary.

Return type
scipy.sparse.dia_matrix

BC_left_extrap()

Sparse matrix to extrapolate solution at left boundary

Returns
Matrix which implements a boundary condition for the left boundary such that the solution at the first two internal grid points is extrapolated to the boundary point.

Return type
scipy.sparse.dia_matrix

BC_left_flat()

Sparse matrix to set Neumann condition at left boundary

Returns
Matrix which implements a boundary condition for the left boundary such that the derivative of the solution is zero at the boundary.

Return type
scipy.sparse.dia_matrix

`BC_left_quad()`

Sparse matrix for quadratic extrapolation at left boundary

Returns
Matrix which implements a boundary condition for the left boundary such that the solution at the first two internal grid points is extrapolated to the boundary point.

Return type
scipy.sparse.dia_matrix

`BC_right_extrap()`

Sparse matrix to extrapolate solution at right boundary

Returns
Matrix which implements a boundary condition for the right boundary such that the solution at the first two internal grid points is extrapolated to the boundary point.

Return type
scipy.sparse.dia_matrix

`centered_difference(y)`

Centered finite difference estimate for dy/dx

Parameters
y (numpy.ndarray) – Vector of values on the grid

Returns
Estimate of the derivative dy/dx constructed using the centered finite difference method

Return type
numpy.ndarray

`ddr()`

Finite difference matrix for (d/dr) f

Returns
Matrix which implements a finite difference approximation to df/dr

Return type
scipy.sparse.dia_matrix

`ddx()`

Finite difference matrix for df/dx (centered)

Returns
Matrix which implements the centered finite difference approximation to df/dx

Return type
scipy.sparse.dia_matrix

`del2()`

Finite difference matrix for d2/dx2

Returns
Matrix which implements a finite difference approximation to (d/dx)(df/dx)

Return type
scipy.sparse.dia_matrix

`del2_radial()`

Finite difference matrix for (1/r)(d/dr)(r (df/dr))

Returns
Matrix which implements a finite difference approximation to (1/r)(d/dr)(r (df/dr))

Return type
scipy.sparse.dia_matrix

`radial_curl()`

Finite difference matrix for (rf)'/r = (1/r)(d/dr)(rf)

Returns
Matrix which implements a finite difference approximation to (rf)'/r = (1/r)(d/dr)(rf)

Return type
scipy.sparse.dia_matrix

`setup_ddx()`

Select between centered and upwind finite difference

Returns
Returns a reference to either *centered_difference()* or *upwind_left()*, based on the configuration option `input_data["method"]`

Return type
function

`upwind_left(y)`

Left upwind finite difference estimate for dy/dx

Parameters
y (numpy.ndarray) – Vector of values on the grid

Returns
Estimate of the derivative dy/dx constructed using the left upwind finite difference method

Return type
numpy.ndarray

`class turbopy.computetools.Interpolators(owner: Simulation, input_data: dict)`

Bases: *ComputeTool*

Interpolate a function y(x) given y at grid points in x

Parameters

- **owner** (Simulation) – The *turbopy.core.Simulation* object that contains this object
- **input_data** (*dict*) – There are no custom configuration options for this tool

`interpolate1D(x, y, kind='linear')`

Given two datasets, return an interpolating function

Parameters

- **x** (*list*) – List of input values to be interpolated
- **y** (*list*) – List of output values to be interpolated
- **kind** (*str*) – Order of function being used to relate the two datasets, defaults to "linear". Passed as a parameter to scipy.interpolate.interpolate.interp1d.

Returns
f – Function which interpolates y(x) given grid *x* and values *y* on the grid.

Return type
scipy.interpolate.interpolate.interp1d

class turbopy.computetools.**PoissonSolver1DRadial**(*owner:* Simulation, *input_data: dict*)

Bases: *ComputeTool*

Solve 1D radial Poisson's Equation, using finite difference methods

Parameters

- **owner** (Simulation) – The *turbopy.core.Simulation* object that contains this object
- **input_data** (*dict*) – There are no custom configuration options for this tool

solve(*sources*)

Solves Poisson's Equation

Parameters
sources (numpy.ndarray) – Vector containing source terms for the Poisson equation

Returns
Vector containing the finite difference solution

Return type
numpy.ndarray

For more details about the turboPy framework, see the published paper in Computer Physics Communications.

Follow along with development at https://github.com/NRL-Plasma-Physics-Division/turbopy

An example of an "app" created with the turboPy framework is available here: https://github.com/NRL-Plasma-Physics-Division/particle-in-field

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W