CoolDwarf

Emily M. Boudreaux

Jun 12, 2024

CONTENTS:

Welcome to the CoolDwarf project. This project aims to provide a easy to use and physically robsut 3D cooling model for fully convective stars.

CoolDwarf is in very early stages of development and is not yet ready for general or scientific use. However, we welcome any feedback or contributions to the project.

The CoolDwarf project is hosted on GitHub at https://github.com/tboudreaux/CoolDwarf

CHAPTER

ONE

DEPEDENICES

CoolDwarf requires the following packages to be installed:

- numpy
- scipy
- matplotlib
- tqdm
- torch

Optional packages:

- cupy

If you are using a CUDA enabled GPU, you can install the cupy package to speed up the calculations significantly. If you do not have a CUDA enabled GPU, you can still use the package, but it will be significantly slower. CoolDwarf will automatically detect if cupy is installed and use it if it is available.

CHAPTER

TWO

INSTALLATION

To install the CoolDwarf package, you can use the following command:

```
git clone https://github.com/tboudreaux/CoolDwarf.git
cd CoolDwarf
pip install .
```

CHAPTER THREE

USAGE

The CoolDwarf package is designed to be easy to use. The primary entry point for using the package is the CoolDwarf.star.VoxelSphere class. This class is used to create a 5D model of a star.

The model is contructed of a grid of equal volume elements spread over a sphere. A MESA model is used to provide the initial temperature and density profiles of the star. An equation of state from Chabrier and Debras 2021 is used to calculate the pressure and energy of the star. The radiative opacity of the star is currentl treated monocromatically and with a simplistic Kramers opacity model (this is a high priority area for improvement).

Evolution of the Cooling model is preformed by calculating the radiative and convective energy gradients at each grid point and to find the new energy after some small time step. The Equation of state is then inverted to update the density, temperature, and pressure of the model.

Timesteps are dynamicall calculated using the Courant-Friedrichs-Lewy (CFL) condition. The CFL condition is used to ensure that the timestep is small enough to prevent the model from becoming unstable. If the user wishes to use a fixed timestep, they can set the cfl_factor to infinity. Alternativley the timestep can be fixed by setting it lower than the CFL condition.

Currently, numerical instabilities exist; however, we are working to resolve these issues. A breif example of how to use the package is shown below (note that neither the EOS tables nor the MESA model are included in the repository; however, these are either readily available online (in the case of the EOS model) or can be generated easily using MESA):

```python
from CoolDwarf.star import VoxelSphere, default_tol
from CoolDwarf.utils import setup_logging
from CoolDwarf.EOS import get_eos
from CoolDwarf.opac import KramerOpac
from CoolDwarf.utils.output import binmod

modelWriter = binmod()

setup_logging(debug=False)

EOS = get_eos("EOS/TABLEEOS_2021_Trho_Y0292_v1", "CD21")
opac = KramerOpac(0.7, 0.02)
sphere = VoxelSphere(
   8e31,
   "BrownDwarfMESA/BD_TEST.mod",
   EOS,
   opac,
   radialResolution=100,
   altitudinalResolition=100,
   azimuthalResolition=100,
```

(continues on next page)

(continued from previous page)

```
    cfl_factor = 0.4,
)
sphere.evolve(maxTime = 60*60*24, pbar=False)
```

3.1 CoolDwarf package

3.1.1 Subpackages

CoolDwarf.EOS package

Subpackages

CoolDwarf.EOS.ChabrierDebras2021 package

Submodules

CoolDwarf.EOS.ChabrierDebras2021.EOS module

EOS.py – EOS class for Chabrier Debras 2021 EOS tables

This module contains the EOS class for the Chabrier Debras 2021 EOS tables. The class is designed to be used with the CoolDwarf Stellar Structure code, and provides the necessary functions to interpolate the EOS tables.

As with all EOS classes in CoolDwarf, the CH21EOS class is designed to accept linear values in cgs units, and return linear values in cgs units.

Dependencies

- pandas
- scipy
- cupy
- torch

Example usage

```
>>> from CoolDwarf.EOS.ChabrierDebras2021.EOS import CH21EOS
>>> eos = CH21EOS("path/to/eos/table")
>>> pressure = eos.pressure(7.0, -2.0)
>>> energy = eos.energy(7.0, -2.0)
```

class CoolDwarf.EOS.ChabrierDebras2021.EOS.**CH21EOS**(*tablePath*)

Bases: `object`

CH21EOS – EOS class for Chabrier Debras 2021 EOS tables

This class is designed to be used with the CoolDwarf Stellar Structure code, and provides the necessary functions to interpolate the Chabrier Debras 2021 EOS tables.

Parameters

tablePath
[str] Path to the Chabrier Debras 2021 EOS table

Attributes

TRange
[tuple] Tuple containing the minimum and maximum temperature (log10(T)) in the EOS table

rhoRange
[tuple] Tuple containing the minimum and maximum density (log10()) in the EOS table

Methods

pressure(logT, logRho)	Interpolates the pressure at the given temperature and density
energy(logT, logRho)	Interpolates the energy at the given temperature and density

`property TRange`

Tuple containing the minimum and maximum temperature (log10(T)) in the EOS table

Returns

tuple
Tuple containing the minimum and maximum temperature (log10(T)) in the EOS table

`check_forward_params`(*logT*, *logRho*)

Check if the given temperature and density are within the bounds of the EOS table. If the values are out of bounds, a ValueError is raised.

Parameters

logT
[float] Log10 of the temperature in K

logRho
[float] Log10 of the density in g/cm^3

Raises

ValueError
If the temperature or density is out of bounds

`energy`(*logT*, *logRho*)

Find the energy at the given temperature and density.

Parameters

logT
[float] Log10 of the temperature in K

logRho
[float] Log10 of the density in g/cm^3

`energy_torch`(*logT*, *logRho*)

Find the energy at the given temperature and density using PyTorch tensors.

Parameters

logT
[torch.Tensor] Log10 of the temperature in K

logRho
[torch.Tensor] Log10 of the density in g/cm^3

`parse_table()`

Parse the Chabrier Debras 2021 EOS table and store the data in the class. Parsing is done using regular expressions to extract the data from the table file. Columns are named according to the table format, and the data is stored in a pandas DataFrame. The temperature and density values are extracted and stored in separate arrays, and the pressure and energy values are interpolated using RegularGridInterpolator.

Column Names (in order): - logT - logP - logRho - logU - logS - dlrho/dlT_P - dlrho/dlP_T - dlS/dlT_P - dlS/dlP_T

The EOS table is stored as a 3D array, with the first dimension corresponding to the temperature, the second dimension corresponding to the density, and the third dimension corresponding to the columns.

The temperature and density arrays are stored as 1D arrays.

`pressure`(*logT*, *logRho*)

Find the pressure at the given temperature and density.

Parameters

logT
[float] Log10 of the temperature in K

logRho
[float] Log10 of the density in g/cm^3

`property rhoRange`

Tuple containing the minimum and maximum density (log10()) in the EOS table

Returns

tuple
Tuple containing the minimum and maximum density (log10()) in the EOS table

Module contents

CoolDwarf.EOS.invert package

Submodules

CoolDwarf.EOS.invert.EOSInverter module

EOSInverter.py – Inverter class for EOS tables

This module contains the Inverter class for EOS tables. The class is designed to be used with the CoolDwarf Stellar Structure code, and provides the necessary functions to invert the EOS tables. Because the inversion problem is non-linear, the Inverter class uses the scipy.optimize.minimize function to find the solution.

Further, because EOSs may not be truley invertible, the Inverter class uses a loss function to find the closest solution to the target energy. over a limited range of temperatures and densities. This is intended to be a range centered around the initial guess for the inversion and limited in size by some expected maximum deviation from the initial guess.

Dependencies

- cupy
- torch
- CoolDwarf.err

Example usage

```
>>> from CoolDwarf.EOS.invert.EOSInverter import Inverter
>>> from CoolDwarf.EOS.ChabrierDebras2021.EOS import CH21EOS
>>> eos = CH21EOS("path/to/eos/table")
>>> inverter = Inverter(eos, TRange, RhoRange)
>>> logTInit, logRhoInit = 7.0, -2.0
>>> newTRange = (6.0, 8.0)
>>> newRhoRange = (-3.0, 0.0)
>>> energy = 1e15
>>> newBounds = (newTRange, newRhoRange)
>>> inverter.set_bounds(newBounds)
>>> logT, logRho = inverter.temperature_density(energy, logTInit, logRhoInit)
```

class `CoolDwarf.EOS.invert.EOSInverter.`**`Inverter`**(*EOS*)

Bases: `object`

Inverter – Inverter class for EOS tables

This class is designed to be used with the CoolDwarf Stellar Structure code, and provides the necessary functions to invert the EOS tables. The Inverter class uses PyTorch optimizers to find the solution to the non-linear inversion problem. Because EOSs may not be truly invertible, the Inverter class uses a loss function to find the closest solution to the target energy over a limited range of temperatures and densities. This is intended to be a range centered around the initial guess for the inversion and limited in size by some expected maximum deviation from the initial guess.

Parameters

EOS
[EOS] EOS object to invert

TRange
[tuple] Tuple containing the minimum and maximum temperature (log10(T)) in the EOS table

RhoRange
[tuple] Tuple containing the minimum and maximum density (log10()) in the EOS table

Attributes

EOS
[EOS] EOS object to invert

_TRange
[tuple] Tuple containing the minimum and maximum temperature (log10(T)) in the EOS table

_RhoRange
[tuple] Tuple containing the minimum and maximum density (log10()) in the EOS table

_bounds
[tuple] Tuple containing the TRange and RhoRange

Methods

temperature_density(energy, logTInit, logRhoInit)	Inverts the EOS to find the temperature and density that gives the target energy
set_bounds(newBounds)	Sets the bounds for the inversion

`set_bounds`(*tRange*, *rRange*)

`temperature_density`(*energy: Tensor*, *logTInit: Tensor*, *logRhoInit: Tensor*, *lr: float = 0.01*, *num_epochs: int = 1000*) → Tensor

`CoolDwarf.EOS.invert.EOSInverter.`**`cupy_to_torch`**(*cupy_array*)

Module contents

Submodules

CoolDwarf.EOS.EOS module

EOS.py – General EOS retreival function for CoolDwarf

This module contains the get_eos function, which is used to retrieve the appropriate EOS object based on the format of the EOS table.

Dependencies

- CoolDwarf.EOS.ChabrierDebras2021.EOS
- CoolDwarf.err

Example usage

```
>>> from CoolDwarf.EOS.EOS import get_eos
>>> eos = get_eos("path/to/eos/table", "CD21")
```

`CoolDwarf.EOS.EOS.`**`get_eos`**(*path: str*, *format: str*)

This function is used to retrieve the appropriate EOS object based on the format of the EOS table. Available formats are: - CD21: Chabrier Debras 2021 EOS tables

Parameters

path
[str] Path to the EOS table

format
[str] Format of the EOS table. Available formats include: CD21 for Chabrier Debras 2021 EOS tables.

Returns

EOS
[EOS] EOS object for the given EOS table

Raises

EOSFormatError
If the format is not recognized

Module contents

CoolDwarf.err package

Submodules

CoolDwarf.err.energy module

exception `CoolDwarf.err.energy.`**`EnergyConservationError`**`(`*msg*`)`

Bases: `Exception`

An exception class for energy conservation error. This exception is raised when the energy conservation is not satisfied during integration of the model.

exception `CoolDwarf.err.energy.`**`NonConvergenceError`**`(`*msg*`)`

Bases: `Exception`

An exception class for non-convergence error. This exception is raised when the solver does not converge.

CoolDwarf.err.eos module

exception `CoolDwarf.err.eos.`**`EOSBoundsError`**`(`*msg*`)`

Bases: `Exception`

An exception class for EOS bounds error. This exception is raised when the bounds for the inversion are not valid.

exception `CoolDwarf.err.eos.`**`EOSFormatError`**`(`*msg*`)`

Bases: `Exception`

An exception class for EOS format error. This exception is raised when the EOS format is not recognized.

exception `CoolDwarf.err.eos.`**`EOSInverterError`**`(`*msg*`)`

Bases: `Exception`

An exception class for EOS inverter error. This exception is raised when an error occurs during the inversion of the EOS.

Module contents

CoolDwarf.ext package

Module contents

CoolDwarf.model package

Subpackages

CoolDwarf.model.dsep package

Submodules

CoolDwarf.model.dsep.dsep module

`CoolDwarf.model.dsep.dsep.`**`parse_dsep_MOD_file`**`()`

Module contents

CoolDwarf.model.mesa package

Submodules

CoolDwarf.model.mesa.mesa module

mesa.py – MESA MOD file parser

This module contains a function to parse MESA MOD files. The function reads the file and extracts the metadata and data sections. Due to the format of the MESA MOD files, the metadata section is read line by line, while the data section is read as a fixed-width file. The data is then stored in a pandas DataFrame. Finally, because MESA uses D instead of E for scientific notation, the function replaces D with E in the data section.

Dependencies

- pandas

Example usage

```
>>> from CoolDwarf.model.mesa.mesa import parse_mesa_MOD_file
>>> df = parse_mesa_MOD_file("path/to/mod/file")
>>> print(df)
```

`CoolDwarf.model.mesa.mesa.`**`parse_mesa_MOD_file`**`(`*`filepath: str`*`)` → DataFrame

This function reads a MESA MOD file and extracts the metadata and data sections. The data is then stored in a pandas DataFrame.

Parameters

filepath
[str] Path to the MESA MOD file

Returns

df
[pd.DataFrame] DataFrame containing the data from the MESA MOD file

Module contents

Submodules

CoolDwarf.model.model module

model.py – General model retreival function for CoolDwarf

This module contains the get_model function, which is used to retrieve the appropriate model object based on the format of the model file.

Dependencies

- pandas
- CoolDwarf.model.mesa
- CoolDwarf.model.dsep

Example usage

```
>>> from CoolDwarf.model.model import get_model
>>> model = get_model("path/to/model/file", "mesa")
>>> print(model)
```

`CoolDwarf.model.model.`**`get_model`**(*path: str*, *format: str*) → DataFrame

This function is used to retrieve the appropriate model object based on the format of the model file. Available formats are: - mesa: MESA MOD files - dsep: DSEP MOD files

Parameters

path
[str] Path to the model file

format
[str] Format of the model file. Available formats include: mesa for MESA MOD files, dsep for DSEP MOD files.

Returns

model
[pd.DataFrame] DataFrame containing the model data

Raises

SSEModelError
If the format is not recognized

Module contents

CoolDwarf.opac package

Subpackages

CoolDwarf.opac.aesopus package

Submodules

CoolDwarf.opac.aesopus.load module

CoolDwarf.opac.aesopus.load.**load_lowtempopac**(*path: str*) → dict

Module contents

Submodules

CoolDwarf.opac.kramer module

karmer.py – Kramer opacity class for CoolDwarf

This module contains the KramerOpac class, which is used to calculate the Kramer opacity for a given temperature and density.

Example usage

```
>>> from CoolDwarf.opac.kramer import KramerOpac
>>> X, Z = 0.7, 0.02
>>> opac = KramerOpac(X, Z)
>>> temp, density = 1e7, 1e-2
>>> kappa = opac.kappa(temp, density)
```

class CoolDwarf.opac.kramer.**KramerOpac**(*X: float*, *Z: float*)

Bases: object

KramerOpac – Kramer opacity class for CoolDwarf

This class is used to calculate the Kramer opacity for a given temperature and density.

Parameters

X
[float] Hydrogen mass fraction

Z
[float] Metal mass fraction

Methods

kappa(temp, density)	Calculates the Kramer opacity at the given temperature and density

kappa(*temp: float*, *density: float*) → float

Function to calculate the Kramer opacity at the given temperature and density

Parameters

temp
[float] Temperature in Kelvin

density
[float] Density in g/cm^3

Returns

kappa
[float] Kramer opacity at the given temperature and density in cm^2/g

CoolDwarf.opac.opacInterp module

`class CoolDwarf.opac.opacInterp.`**`OPACInterp`**(*opacDict*, *X*, *Z*)

Bases: `object`

Methods

kappa

kappa(*temp*, *density*)

Module contents

CoolDwarf.star package

Submodules

CoolDwarf.star.sphere module

sphere.py

This module contains the VoxelSphere class, which represents a 3D voxelized sphere model for a star. The VoxelSphere class provides methods for computing various physical properties of the star, including its radius, enclosed mass, energy flux, and more.

The VoxelSphere class uses an equation of state (EOS) for the star, which can be inverted to compute pressure and temperature grids. It also provides methods for updating the energy of the star and recomputing its state.

The module imports several utility functions and constants from the CoolDwarf package, as well as classes for handling errors related to energy conservation and non-convergence.

All units are in non log space cgs unless otherwise specified.

Dependencies

- numpy
- tqdm
- torch
- cupy
- pandas
- scipy.interpolate
- CoolDwarf.utils.math
- CoolDwarf.utils.const
- CoolDwarf.utils.format
- CoolDwarf.EOS
- CoolDwarf.model
- CoolDwarf.err

Classes

- VoxelSphere: Represents a 3D voxelized sphere model for a star.

Functions

- default_tol: Returns a dictionary of default numerical tolerances for the cooling model.

Exceptions

- EnergyConservationError: Raised when there is a violation of energy conservation.
- NonConvergenceError: Raised when a computation fails to converge.
- VolumeError: Raised when the volume error is greater than the tolerance.
- ResolutionError: Raised when the resolution is insufficient.

Example Usage

```
>>> from CoolDwarf.star import VoxelSphere
>>> from CoolDwarf.utils.plot import plot_3d_gradients, visualize_scalar_field
>>> from CoolDwarf.utils import setup_logging
>>> from CoolDwarf.EOS import get_eos
>>> from CoolDwarf.opac import KramerOpac
>>> from CoolDwarf.EOS.invert import Inverter
```

```
>>> import numpy as cp
>>> import matplotlib.pyplot as plt
>>> setup_logging(debug=True)
```

```
>>> EOS = get_eos("EOS/TABLEEOS_2021_Trho_Y0292_v1", "CD21")
>>> opac = KramerOpac(0.7, 0.02)
>>> sphere = VoxelSphere(8e31, "BrownDwarfMESA/BD_TEST.mod", EOS, opac,␣
↪radialResolution=50, altitudinalResolition=10, azimuthalResolition=20)
>>> sphere.evolve(maxTime=3.154e+7, dt=86400)
>>> print(f"Surface Temp: {sphere.surface_temperature_profile}")
```

class `CoolDwarf.star.sphere.`**`VoxelSphere`**(*mass*, *model*, *EOS*, *opac*, *pressureRegularization=1e-05*, *radialResolution=10*, *azimuthalResolition=10*, *altitudinalResolition=10*, *t0=0*, *X=0.75*, *Y=0.25*, *Z=0*, *tol={'maxEChange': 0.0001, 'relax': 1e-06, 'volCheck': 0.01}*, *modelFormat='mesa'*, *alpha=1.901*, *mindt=0.1*, *cfl_factor=0.5*, *imodelOut=False*, *imodelOutCadence=1000*, *imodelOutCadenceUnit='s'*, *fmodelOut=True*)

Bases: `object`

A class to represent a 3D voxelized sphere model for a star.

Raises

EnergyConservationError
If there is a violation of energy conservation.

NonConvergenceError
If a computation fails to converge.

Attributes

CONST
[dict] A dictionary of physical constants.

mass
[float] Returns the mass grid for the star.

model
[str] The name of the stellar model to use.

modelFormat
[str] The format of the stellar model. Default is mesa. May also be dsep.

EOS
[EOS] The equation of state for the star.

opac
[Opac] The opacity of the star.

pressureRegularization
[float] A regularization parameter for the pressure computation.

radialResolution
[int] The number of radial divisions in the voxelized sphere.

azimuthalResolution
[int] The number of azimuthal divisions in the voxelized sphere.

t0
[float] The initial time of the star.

X
[float] The hydrogen mass fraction of the star.

Y
[float] The helium mass fraction of the star.

Z
[float] The metal mass fraction of the star.

alpha
[float] The mixing length parameter.

mindt
[float] The minimum timestep for the star.

cfl_factor
[float] The Courant-Friedrichs-Lewy factor for the star.

tol
[dict] A dictionary of numerical tolerances for the cooling model. Keys are relax and maxEChange. Default is {relax: 1e-6, maxEChange: 1e-4, volCheck: 1e-2}. Relax is the relaxation parameter for the energy update. MaxEChange is the maximum fractional change in energy allowed per timestep. volCheck is the maximum fractional error in volume allowed.

Methods

Cp([delta_t])	Computes the specific heat capacity of the star at constant pressure.
as_dict()	Returns a dictionary representation of the star. Returns ------- dict: A dictionary representation of the star.
as_pandas()	Returns a pandas DataFrame representation of the star. Returns ------- pandas.DataFrame: A DataFrame representation of the star.
evolve([maxTime, dt, pbar])	Evolves the star over a specified time period using a specified timestep.
save(filename)	Save to a binary file format.
spherical_grid_equal_volume(numRadial, ...)	Generate points within a sphere with equal volume using a stratified sampling approach.
timestep([userdt])	Computes a timestep for the star based on the CFL condition or the user-specified timestep.

CONST = {'G': 6.6743e-08, 'a': 7.5646e-15, 'c': 29979245800.0, 'mH': 1.00784, 'mHe': 4.002602}

Cp(*delta_t: float = 1*)

Computes the specific heat capacity of the star at constant pressure.

Parameters

delta_t
[float, optional] A small change in temperature for computing the specific heat capacity. Default is 1e-5.

Returns

xp.ndarray: The specific heat capacity of the star at constant pressure.

`as_dict()`

Returns a dictionary representation of the star. Returns -

dict: A dictionary representation of the star.

`as_pandas()`

Returns a pandas DataFrame representation of the star. Returns -

pandas.DataFrame: A DataFrame representation of the star.

`property cfl_dt: float`

Computes the timestep based on the Courant-Friedrichs-Lewy (CFL) condition.

Returns

float: The timestep based on the CFL condition.

`property convective_energy_flux: Tuple[ndarray, ndarray, ndarray]`

Computes the convective energy flux in the radial, azimuthal, and altitudinal directions. We take a mixing length theory approach to compute the convective energy flux.

The convective enertgy flux for some coordinate direction is given by: Fconv = (1/2) * rho * Cp * v * (TGrad - ad) where rho is the density, Cp is the specific heat capacity, v is the convective velocity along that coordinate axis, TGrad is the temperature gradient along that coordinte axis, and ad is the adiabatic gradient.

Returns

tuple: A tuple of 3D arrays representing the convective energy flux in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (FconvR, FconvTheta, FconvPhi).

`property convective_overturn_timescale: Tuple[ndarray, ndarray, ndarray]`

Computes the convective overturn timescale in the radial, azimuthal, and altitudinal directions. The convective overturn timescale is given by the mixing length divided by the convective velocity.

Returns

tuple: A tuple of 3D arrays representing the convective overturn timescale in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (tauR, tauTheta, tauPhi).

`property convective_velocity: Tuple[ndarray, ndarray, ndarray]`

Computes the convective velocity in the radial, azimuthal, and altitudinal directions. The convective velocity is given by the mixing length divided by two times the square root of the product of the gravitational acceleration and the difference between the adiabatic gradient and the temperature gradient.

Returns

tuple: A tuple of 3D arrays representing the convective velocity in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (vR, vTheta, vPhi).

`property dEdt: ndarray`

Computes the time derivative of the energy for the star.

Returns

xp.ndarray: The time derivative of the energy for the star.

`property density: ndarray`

Returns the density grid for the star. Returns -

xp.ndarray: The density grid for the star.

`property enclosed_mass`

Computes the enclosed mass of the star as a function of radius.

Returns

interp1d: A 1D interpolation function for the enclosed mass.

`property energy: ndarray`

Returns the energy grid for the star.

`property energy_flux: Tuple[Tuple[ndarray, ndarray, ndarray], Tuple[ndarray, ndarray, ndarray]]`

Computes the energy flux in the radial, azimuthal, and altitudinal directions.

Returns

tuple: A tuple of 3D arrays representing the energy flux in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of ((fluxR, fluxTheta, fluxPhi), (fluxR, fluxTheta, fluxPhi)).

`evolve`(*maxTime: float = 31540000.0*, *dt: float = 86400*, *pbar=False*)

Evolves the star over a specified time period using a specified timestep.

Parameters

maxTime
[float, optional] The maximum time to evolve the star. Default is 3.154e+7.

dt
[float, optional] The timestep to use for the evolution. Default is 86400.

pbar
[bool, optional] Display a progress bar for the evolution. Default is False.

`property flux_divergence: Tuple[Tuple[ndarray, ndarray, ndarray], Tuple[ndarray, ndarray, ndarray]]`

Computes the divergence of the energy flux in the radial, azimuthal, and altitudinal directions.

Returns

tuple: A tuple of 3D arrays representing the divergence of the energy flux in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of ((delFConvR, delFConvTheta, delFConvPhi), (delFRadR, delFRadTheta, delFRadPhi)).

`property gradRadEr: Tuple[ndarray, ndarray, ndarray]`

Computes the radiative energy gradient.

Returns

tuple: A tuple of 3D arrays representing the radiative energy gradient in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (delErR, delErTheta, delErPhi).

`property gradT: Tuple[ndarray, ndarray, ndarray]`

Computes the temperature gradients in the radial, azimuthal, and altitudinal directions.

Returns

tuple: A tuple of 3D arrays representing the temperature gradients in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (tGradR, tGradTheta, tGradPhi).

`property gravitational_acceleration: ndarray`

Computes the gravitational acceleration for the star. If the mass grid is zero at a given grid point, the gravitational acceleration is set to infinity to deal with the singularity at r=0.

Returns

xp.ndarray: The gravitational acceleration for the star.

`property mass: ndarray`

Returns the mass grid for the star. Returns -

xp.ndarray: The mass grid for the star.

`property mixing_length: ndarray`

Computes the mixing length for the star.

Returns

xp.ndarray: The mixing length for the star.

`property pressure: ndarray`

Returns the pressure grid for the star. Returns -

xp.ndarray: The pressure grid for the star.

`property pressure_scale_height: ndarray`

Computes the pressure scale height for the star.

Returns

xp.ndarray: The pressure scale height for the star.

`property radiative_energy_flux: Tuple[ndarray, ndarray, ndarray]`

Computes the radiative energy flux in the radial, azimuthal, and altitudinal directions.

Returns

tuple: A tuple of 3D arrays representing the radiative energy flux in the radial, azimuthal, and altitudinal directions.
The arrays are in the form of (fluxRadR, fluxRadTheta, fluxRadPhi).

`property radius`

Returns the radius of the star.

Returns

float: The radius of the star.

`save`(*filename: str*) → bool

Save to a binary file format. Currently the model format is being defined in the joplin notebook I am using to keep track of development.

Parameters

filename
[str] The filename to save the star to.

Returns

bool: A flag indicating if the save was successful.

`spherical_grid_equal_volume`(*numRadial*, *numTheta*, *numPhi*, *radius*)

Generate points within a sphere with equal volume using a stratified sampling approach. Returns radius, theta, phi as meshgrids and volume elements for each point.

Parameters

numRadial
[int] Number of radial segments.

numTheta
[int] Number of azimuthal segments.

numPhi
[int] Number of altitudinal segments.

radius
[float] Radius of the sphere.

Returns

tuple: A tuple of meshgrids for the radial, azimuthal, and altitudinal positions, and the volume elements.

Raises

VolumeError
If the volume error is greater than the tolerance.

`property surface_temperature_profile: ndarray`

Computes the surface temperature profile for the star.

Returns

xp.ndarray: The surface temperature profile for the star.

`property temperature: ndarray`

Returns the temperature grid for the star.

Returns

xp.ndarray: The temperature grid for the star.

`timestep`(*userdt: float = inf*) → float

Computes a timestep for the star based on the CFL condition or the user-specified timestep. The actual timestep used is the minimum of the CFL timestep and the user-specified timestep, and this will be returned.

The energy of the star is then updated based on the computed timestep. Following this, the energy is used to invert the EOS and update the temperature and density grids. The energy conservation is checked, and if the energy change is greater than the maximum energy change tolerance, the timestep is halved and the energy, temperature, density, and pressure grids are reset to their initial values.

When the energy conservation is satisfied, the evolutionary step is incremented, and the time is updated based on the the acutal timestep used. The timestep is then returned.

Parameters

userdt
[float, optional] The user-specified timestep. Default is xp.inf.

Returns

float: The actual timestep used for the star.

Raises

EnergyConservationError
If there is a violation of energy conservation.

NonConvergenceError
If the model fails to converge after a certain number of timesteps.

Examples

```
>>> star = VoxelSphere(...)
>>> star.timestep()
```

`CoolDwarf.star.sphere.`**`default_tol`**`()`

Returns a dictionary of default numerical tolerances for the cooling model.

Returns

**dict: A dictionary of default numerical tolerances for the cooling model.
Keys are relax, maxEChange, and volCheck.**

Module contents

CoolDwarf.utils package

Subpackages

CoolDwarf.utils.const package

Submodules

CoolDwarf.utils.const.const module

const.py – Constants for CoolDwarf

This module contains the physical constants used in CoolDwarf.

Constants include: - mH: Hydrogen atomic mass in amu (1.00784) - mHe: Helium atomic mass in amu (4.002602) - c: Speed of light in cgs units (2.99792458e10) - a: Radiation constant in cgs units (7.5646e-15) - G: Gravitational constant in cgs units (6.6743e-8)

Example usage

```
>>> from CoolDwarf.utils.const.const import CONST
>>> print(CONST['mH'])
```

Module contents

CoolDwarf.utils.format package

Submodules

CoolDwarf.utils.format.format module

`CoolDwarf.utils.format.format.`**`format_number`**(*x*, *max_width*)

`CoolDwarf.utils.format.format.`**`pretty_print_3d_array`**(*array3D*, *mask*, *decimals=3*)

Module contents

CoolDwarf.utils.interp package

Submodules

CoolDwarf.utils.interp.interpolate module

`CoolDwarf.utils.interp.interpolate.`**`find_closest_values`**(*numList*, *targetValue*)

`CoolDwarf.utils.interp.interpolate.`**`linear_interpolate_dataframes`**(*df_dict*, *target_key*)

`CoolDwarf.utils.interp.interpolate.`**`linear_interpolate_ndarray`**(*arrays*, *keys*, *target*)

Module contents

CoolDwarf.utils.math package

Submodules

CoolDwarf.utils.math.calc module

calc.py – Calculus related functions for CoolDwarf

This module contains functions related to calculus that are used in CoolDwarf.

Functions include: - partial_derivative_x: Function to calculate the partial derivative along the x-axis - compute_partial_derivatives: Function to compute the partial derivatives of a scalar field

Dependencies

- numpy

Example usage

```
>>> import numpy as np
>>> from CoolDwarf.utils.math.calc import partial_derivative_x, compute_partial_
↪derivatives
>>> var = np.random.rand(10, 10, 10)
>>> dx = 1.0
>>> partial_x = partial_derivative_x(var, dx)
>>> x = np.linspace(0, 1, 10)
>>> y = np.linspace(0, 1, 10)
>>> z = np.linspace(0, 1, 10)
>>> dfdx, dfdy, dfdz = compute_partial_derivatives(var, x, y, z)
```

`CoolDwarf.utils.math.calc.`**`compute_partial_derivatives`**`(`*scalar_field: ndarray*, *x: ndarray*, *y: ndarray*, *z: ndarray*`)` → Tuple[ndarray, ndarray, ndarray]

Function to compute the partial derivatives of a scalar field

Parameters

scalar_field
[np.ndarray] 3D array representing the scalar field

x
[np.ndarray] Array of x-axis values

y
[np.ndarray] Array of y-axis values

z
[np.ndarray] Array of z-axis values

Returns

dfdx
[np.ndarray] Array containing the partial derivative along the x-axis

dfdy
[np.ndarray] Array containing the partial derivative along the y-axis

dfdz
[np.ndarray] Array containing the partial derivative along the z-axis

`CoolDwarf.utils.math.calc.`**`partial_derivative_x`**`(`*var: ndarray*, *dx: float*`)` → ndarray

Function to calculate the partial derivative along the x-axis

Parameters

var
[np.ndarray] Array of values to calculate the partial derivative of

dx
[float] Spacing between the x-axis points

Returns

partial_x
[np.ndarray] Array containing the partial derivative along the x-axis

CoolDwarf.utils.math.kernel module

`CoolDwarf.utils.math.kernel.`**`make_3d_kernels`**`()`

Module contents

CoolDwarf.utils.misc package

Submodules

CoolDwarf.utils.misc.evolve module

CoolDwarf.utils.misc.logging module

logging.py – Logging setup for CoolDwarf

This module contains the setup_logging function, which is used to set up the logging configuration for CoolDwarf.

Example usage

```
>>> from CoolDwarf.utils.misc.logging import setup_logging
>>> setup_logging(debug=True)
```

`class CoolDwarf.utils.misc.logging.`**`CustomFilter`**`(`*name=''*`)`

Bases: `Filter`

Methods

`filter`(record)	Determine if the specified record is to be logged.

`filter``(`*record*`)`

Determine if the specified record is to be logged.

Returns True if the record should be logged, or False otherwise. If deemed appropriate, the record may be modified in-place.

`class CoolDwarf.utils.misc.logging.`**`ExcludeCustomFilter`**`(`*name=''*`)`

Bases: `Filter`

Methods

filter(record)	Determine if the specified record is to be logged.

filter(*record*)

Determine if the specified record is to be logged.

Returns True if the record should be logged, or False otherwise. If deemed appropriate, the record may be modified in-place.

CoolDwarf.utils.misc.logging.**evolve_log**(*self*, *message*, **args*, ***kwargs*)

CoolDwarf.utils.misc.logging.**setup_logging**(*debug: bool = False*)

This function is used to set up the logging configuration for CoolDwarf.

Parameters

debug
[bool, default=False] If True, sets the logging level to DEBUG. Otherwise, sets the logging level to INFO.

CoolDwarf.utils.misc.ndarray module

class CoolDwarf.utils.misc.ndarray.**CallbackNDArray**(*input_array*, *callback=None*, **args*, ***kwargs*)

Bases: ndarray

Attributes

T
View of the transposed array.

base
Base object if memory is from some other object.

ctypes
An object to simplify the interaction of the array with the ctypes module.

data
Python buffer object pointing to the start of the arrays data.

dtype
Data-type of the arrays elements.

flags
Information about the memory layout of the array.

flat
A 1-D iterator over the array.

imag
The imaginary part of the array.

itemsize
Length of one array element in bytes.

nbytes
Total bytes consumed by the elements of the array.

ndim
Number of array dimensions.

real
The real part of the array.

shape
Tuple of array dimensions.

size
Number of elements in the array.

strides
Tuple of bytes to step in each dimension when traversing an array.

Methods

`all`([axis, out, keepdims, where])	Returns True if all elements evaluate to True.
`any`([axis, out, keepdims, where])	Returns True if any of the elements of *a* evaluate to True.
`argmax`([axis, out, keepdims])	Return indices of the maximum values along the given axis.
`argmin`([axis, out, keepdims])	Return indices of the minimum values along the given axis.
`argpartition`(kth[, axis, kind, order])	Returns the indices that would partition this array.
`argsort`([axis, kind, order])	Returns the indices that would sort this array.
`astype`(dtype[, order, casting, subok, copy])	Copy of the array, cast to a specified type.
`byteswap`([inplace])	Swap the bytes of the array elements
`choose`(choices[, out, mode])	Use an index array to construct a new array from a set of choices.
`clip`([min, max, out])	Return an array whose values are limited to `[min, max]`.
`compress`(condition[, axis, out])	Return selected slices of this array along given axis.
`conj`()	Complex-conjugate all elements.
`conjugate`()	Return the complex conjugate, element-wise.
`copy`([order])	Return a copy of the array.
`cumprod`([axis, dtype, out])	Return the cumulative product of the elements along the given axis.
`cumsum`([axis, dtype, out])	Return the cumulative sum of the elements along the given axis.
`diagonal`([offset, axis1, axis2])	Return specified diagonals.
`dump`(file)	Dump a pickle of the array to the specified file.
`dumps`()	Returns the pickle of the array as a string.
`fill`(value)	Fill the array with a scalar value.
`flatten`([order])	Return a copy of the array collapsed into one dimension.
`getfield`(dtype[, offset])	Returns a field of the given array as a certain type.
`item`(*args)	Copy an element of an array to a standard Python scalar and return it.
`itemset`(*args)	Insert scalar into an array (scalar is cast to array's dtype, if possible)
`max`([axis, out, keepdims, initial, where])	Return the maximum along a given axis.

continues on next page

Table 1 – continued from previous page

`mean`([axis, dtype, out, keepdims, where])	Returns the average of the array elements along given axis.
`min`([axis, out, keepdims, initial, where])	Return the minimum along a given axis.
`newbyteorder`([new_order])	Return the array with the same data viewed with a different byte order.
`nonzero`()	Return the indices of the elements that are non-zero.
`partition`(kth[, axis, kind, order])	Rearranges the elements in the array in such a way that the value of the element in kth position is in the position it would be in a sorted array.
`prod`([axis, dtype, out, keepdims, initial, ...])	Return the product of the array elements over the given axis
`ptp`([axis, out, keepdims])	Peak to peak (maximum - minimum) value along a given axis.
`put`(indices, values[, mode])	Set `a.flat[n] = values[n]` for all *n* in indices.
`ravel`([order])	Return a flattened array.
`repeat`(repeats[, axis])	Repeat elements of an array.
`reshape`(shape[, order])	Returns an array containing the same data with a new shape.
`resize`(new_shape[, refcheck])	Change shape and size of array in-place.
`round`([decimals, out])	Return *a* with each element rounded to the given number of decimals.
`searchsorted`(v[, side, sorter])	Find indices where elements of v should be inserted in a to maintain order.
`setfield`(val, dtype[, offset])	Put a value into a specified place in a field defined by a data-type.
`setflags`([write, align, uic])	Set array flags WRITEABLE, ALIGNED, WRITEBACKIFCOPY, respectively.
`sort`([axis, kind, order])	Sort an array in-place.
`squeeze`([axis])	Remove axes of length one from *a*.
`std`([axis, dtype, out, ddof, keepdims, where])	Returns the standard deviation of the array elements along given axis.
`sum`([axis, dtype, out, keepdims, initial, where])	Return the sum of the array elements over the given axis.
`swapaxes`(axis1, axis2)	Return a view of the array with *axis1* and *axis2* interchanged.
`take`(indices[, axis, out, mode])	Return an array formed from the elements of *a* at the given indices.
`tobytes`([order])	Construct Python bytes containing the raw data bytes in the array.
`tofile`(fid[, sep, format])	Write array to a file as text or binary (default).
`tolist`()	Return the array as an `a.ndim`-levels deep nested list of Python scalars.
`tostring`([order])	A compatibility alias for *tobytes*, with exactly the same behavior.
`trace`([offset, axis1, axis2, dtype, out])	Return the sum along diagonals of the array.
`transpose`(*axes)	Returns a view of the array with axes transposed.
`var`([axis, dtype, out, ddof, keepdims, where])	Returns the variance of the array elements, along given axis.
`view`([dtype][, type])	New view of array with the same data.

dot

Module contents

CoolDwarf.utils.plot package

Submodules

CoolDwarf.utils.plot.plot2d module

`CoolDwarf.utils.plot.plot2d.`**`plot_polar_slice`**(*sphere*, *data*, *phi_slice=0*, *theta_offset=0*, *fname='polar_slice.png'*)

`CoolDwarf.utils.plot.plot2d.`**`visualize_scalar_field`**(*scalar_field*, *slice_axis='z'*, *slice_index=None*, ***kwargs*)

CoolDwarf.utils.plot.plot3d module

`CoolDwarf.utils.plot.plot3d.`**`plot_3d_gradients`**(*grid_points*, *gradients*, *radius*, *sphere_radius=1*, *cell=False*)

Module contents

Module contents

3.1.2 Module contents

3.2 Indices and tables

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX