

---

# **hdrdrPy**

***Release 0.0.1***

**Anthony Arfaux**

**Jan 09, 2026**



# CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
<b>3</b>	<b>Summary</b>	<b>5</b>
3.1	Examples . . . . .	5
3.2	htrdrPy documentation . . . . .	5
3.2.1	htrdrPy.data module . . . . .	5
3.2.2	htrdrPy.geometry module . . . . .	20
3.2.3	htrdrPy.script module . . . . .	26
3.2.4	htrdrPy.postprocess module . . . . .	28
3.2.5	htrdrPy.helperFunctions module . . . . .	32
	<b>Python Module Index</b>	<b>35</b>
	<b>Index</b>	<b>37</b>



## **INTRODUCTION**

htrdrPy is a wrapper for htrdr-planets (<https://www.meso-star.com/projects/htrdr/htrdr.html>). htrdr-planets is not included in the wrapper, it must be install separately. htrdrPy is meant to simplify the use, by managing the input data, scripts, observation geometry and traitement of the results.



## INSTALLATION

htdrPy can be installed from the Python Package Index with the following command:

```
$ pip install htldrPy
```





## SUMMARY

### 3.1 Examples

This is an example of use of the htrdrPy package

### 3.2 htrdrPy documentation

#### 3.2.1 htrdrPy.data module

**class** htrdrPy.data.Data(radius, nTheta=None, nPhi=None, mass=None, gravity=None, name="")

Bases: object

htrdrPy.Data is a class aiming to handle the optical and physical properties of the system and to create the input files for htrdr.

#### Examples

The first step is the creation of a instance of htrdrPy.Data:

```
>>> d = htrdrPy.Data(radius=1e6, nTheta=30, nPhi=50, name="Planet")
```

The next step is to provide the physical and radiative properties of the atmosphere and ground. Different methods exist depending on the case considered. In the following, we consider a 1D set of data forming an horizontally homogeneous planet.

```
>>> nLevel = 50
>>> nCoeff = 4
>>> nWavelengths = 20
>>> weights = np.array(nWavelengths * [0.2, 0.3, 0.3, 0.2]).reshape(nWavelengths,
↪ nCoeff)
>>> altitudes = np.linspace(0, 5e5, nLevel)
>>> temperatures = np.linspace(300, 500, nLevel)
>>> scatt = np.linspace(1e-8, 1e-2,
... nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> absor = np.linspace(1e-5, 1e-1,
... nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> asymm = np.linspace(0, 1,
... nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:] = bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> surfTemp= 300
```

(continues on next page)

(continued from previous page)

```

>>> surfAlb= np.ones(nWavelengths) * 0.5
>>> d.makeAtmosphereFrom1D({
...     "nLevel": nLevel,
...     "nCcoeff": nCcoeff,
...     "nWavelengths": nWavelengths,
...     "weights": weights,
...     "altitude (m)": altitudes,
...     "temperature (K)": temperatures,
...     "scattering (m-1)": scatt,
...     "absorption (m-1)": absor,
...     "assymetry": asymm,
...     "wavelength": wavelengths,
...     "band low": bandsLow,
...     "band up": bandsUp
... })
Mesh generator. Ntheta = 30, Nphi = 50, Nz = 50, r_min = 1000000.0, r_max =
↳ 1500000.0
Generating points...
Generating nodes...
Hexahedron & Octahedron generation completed. N_Hexahedron = 4802, N_Octahedron
↳ = 64827
Generating Tetrahedrons...
Assigning data to nodes ...
>>> d.makeGroundFrom1D(surfTemp, {
...     "kind": "lambertian",
...     "albedo": surfAlb,
...     "bands": np.array([bandsLow, bandsUp]).T
... })
Mesh generator. Ntheta = 30, Nphi = 50, R = 1000000.0
Generating points...
Generating nodes...
triangles,rectangles generation completed. N_triangles = 98, N_rectangles = 1323
Generating triangles...
generating bin...

```

Once the data have been provided, the method `htrdrPy.Data.writeInputs` will generate the input file in an `input_{name}` folder.

```

>>> d.writeInputs()
generating surface mesh bin file...
Nnodes = 5586 Ncells = 2744 dim_node = 3 dim_cell = 3
bin generation completed.
generating surface properties bin file...
bin generation completed.
generating atmosphere mesh bin file...
Nnodes = 547428 Ncells = 338541 dim_node = 3 dim_cell = 4
bin generation completed.
generating gas temperature bin file...
bin generation completed.
generating gas properties bin file...
100%|_____
↳
↳ 20/20 [00:11<00:00, 1.69it/s]
bin generation completed.
generating haze properties bin file...
100%|_____

```

(continues on next page)

(continued from previous page)

```

→ 20/20 [00:04<00:00, 4.95it/s]
bin generation completed.
generating haze phase function bin file...
bin generation completed.

```

### Parameters

- **radius** (*float*) – Radius of the planet [m].
- **nTheta** (*int, optional, not requested if meshes loaded from file*) – Number of latitude points in the range  $[0^\circ, 360^\circ]$  to be used.
- **nPhi** (*int, optional, not requested if meshes loaded from file*) – Number of longitude points in the range  $[-180^\circ, 180^\circ]$ .
- **gravity** (*float, optional,*) – Gravity of the planet  $[m/s^2]$ . Requested only for LMDZ-PCM input/output files to extract the altitude grid from the geopotential. Alternatively, the user can provide the planet mass.
- **mass** (*float, optional,*) – Mass of the planet [kg]. Requested only for LMDZ-PCM input/output files to extract the altitude grid from the geopotential. Alternatively, the user can provide the planet gravity.
- **name** (*str, optional, default uses a counter of the number of instances of htrdrPy.Data*) – Name for the dataset, which will be used to name the input folders.

### cleanInputs()

Remove the inputs\_{name} folder.

### makeAtmosphereFrom1D(data)

Generate a spherical atmosphere from single column data. The data are provided at the levels, i.e. at the interface between layers.

#### Parameters

**data** (*dict*) – Dictionnary with the following items:

- **"nLevel"**  
[int] Number of levels.
- **"nCoeff": int**  
Number of quadrature points for the k-coeff.
- **"nWavelengths": int**  
Number of wavelengths.
- **"nAngle": int, optional**  
Number of angles in the phase functions (not required if using the builtin Henyey-Greenstein phase function).
- **"weights": numpy.ndarray**  
The weights of the nCoeff quadrature points (shape=(nWavelengths, nCoeff)).
- **"altitude (m)": numpy.ndarray**  
Array of altitudes (shape=(nLevel), [m]).
- **"temperature (K)": numpy.ndarray**  
Array of temperatures (shape=(nLevel), [K]).
- **"scattering (m-1)": numpy.ndarray**  
Array of scattering coefficients (shape=(nWavelengths, nLevel, nCoeff), [m-1]).
- **"absorption (m-1)": numpy.ndarray**  
Array of absorption coefficients (shape=(nWavelengths, nLevel, nCoeff), [m-1]).

•**"angles (°)": numpy.ndarray, optional**

Array of angle values for the phase function (shape=(nAngle), float [°]). Only required when providing the phaseFunc.

•**"phaseFunc": numpy.ndarray, optional**

Array of discrete phase functions (shape = (nWavelengths, nLevel, nAngle, nCoeff)). Alternatively, the user can provide an array of asymmetry parameter.

•**"asymmetry": numpy.ndarray, optional**

Array of asymmetry parameter (shape=(nWavelengths, nLevel, nCoeff)). Alternatively, the user can provide the discrete phase function.

•**"wavelength": numpy.ndarray, optional**

Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the band center. The value is used of the phase function.

•**"band low": numpy.ndarray**

Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

•**"band up": numpy.ndarray**

Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

## Notes

The nTheta and nPhi parameters are used to define the resolution of the atmosphere mesh and are therefore mandatory.

## Examples

```
>>> d = htrdrPy.Data(radius=1e6, nTheta=30, nPhi=50, name="Planet")
>>> nLevel = 50
>>> nCoeff = 4
>>> nWavelengths = 20
>>> weights = np.array(nWavelengths * [0.2, 0.3, 0.3, 0.2]).
↳ reshape(nWavelengths, nCoeff)
>>> altitudes = np.linspace(0, 5e5, nLevel)
>>> temperatures = np.linspace(300, 500, nLevel)
>>> scatt = np.linspace(1e-8, 1e-2,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> absor = np.linspace(1e-5, 1e-1,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> asymm = np.linspace(0, 1,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:] = bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> d.makeAtmosphereFrom1D({
...     "nLevel": nLevel,
...     "nCoeff": nCoeff,
...     "nWavelengths": nWavelengths,
...     "weights": weights,
...     "altitude (m)": altitudes,
...     "temperature (K)": temperatures,
...     "scattering (m-1)": scatt,
...     "absorption (m-1)": absor,
```

(continues on next page)

(continued from previous page)

```

...     "asymmetry": asymm,
...     "wavelength": wavelengths,
...     "band low": bandsLow,
...     "band up": bandsUp
...     })
Mesh generator. Ntheta = 30, Nphi = 50, Nz = 50, r_min = 1000000.0, r_max = 1500000.0
Generating points...
Generating nodes...
Hexahedron & Octahedron generation completed. N_Hexahedron = 4802, N_Octahedron = 64827
Generating Tetrahedrons...
Assigning data to nodes ...

```

**makeAtmosphereFrom1D\_PP(*data*)**

Generate a plan parallel atmosphere from single column data. The data are provided at the levels, i.e. at the interface between layers.

**Parameters**

**data** (*dict*) – Dictionary with the following items:

- **"nLevel"**  
[int] Number of levels.
- **"nCoeff": int**  
Number of quadrature points for the k-coeff.
- **"nWavelengths": int**  
Number of wavelengths.
- **"nAngle": int, optional**  
Number of angles in the phase functions (not required if using the builtin Henyey-Greenstein phase function).
- **"weights": numpy.ndarray**  
The weights of the nCoeff quadrature points (shape=(nWavelengths, nCoeff)).
- **"altitude (m)": numpy.ndarray**  
Array of altitudes (shape=(nLevel), [m]).
- **"temperature (K)": numpy.ndarray**  
Array of temperatures (shape=(nLevel), [K]).
- **"scattering (m-1)": numpy.ndarray**  
Array of scattering coefficients (shape=(nWavelengths, nLevel, nCoeff), [m-1]).
- **"absorption (m-1)": numpy.ndarray**  
Array of absorption coefficients (shape=(nWavelengths, nLevel, nCoeff), [m-1]).
- **"angles (°)": numpy.ndarray, optional**  
Array of angle values for the phase function (shape=(nAngle), float [°]). Only required when providing the phaseFunc.
- **"phaseFunc": numpy.ndarray, optional**  
Array of discrete phase functions (shape = (nWavelengths, nLevel, nAngle, nCoeff)). Alternatively, the user can provide an array of asymmetry parameter.
- **"asymmetry": numpy.ndarray, optional**  
Array of asymmetry parameter (shape=(nWavelengths, nLevel, nCoeff)). Alternatively, the user can provide the discrete phase function.
- **"wavelength": numpy.ndarray, optional**  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the band center. The value is used of the phase function.

•**"band low": numpy.ndarray**

Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

•**"band up": numpy.ndarray**

Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

## Notes

In plan-parallel mode, the radius parameter passed at the initialisation of the instance is used as the horizontal expansion of the ground. Make sure to use a large enough value. Also, the nTheta and nPhi parameters are not used.

## Examples

```
>>> d = htrdrPy.Data(radius=1e6, nTheta=30, nPhi=50, name="Planet")
>>> nLevel = 50
>>> nCoeff = 4
>>> nWavelengths = 20
>>> weights = np.array(nWavelengths * [0.2, 0.3, 0.3, 0.2]).
↳ reshape(nWavelengths, nCoeff)
>>> altitudes = np.linspace(0, 5e5, nLevel)
>>> temperatures = np.linspace(300, 500, nLevel)
>>> scatt = np.linspace(1e-8, 1e-2,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> absor = np.linspace(1e-5, 1e-1,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> asymm = np.linspace(0, 1,
...     nLevel*nCoeff*nWavelengths).reshape((nWavelengths, nLevel, nCoeff))
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:] = bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> d.makeAtmosphereFrom1D_PP({
...     "nLevel": nLevel,
...     "nCoeff": nCoeff,
...     "nWavelengths": nWavelengths,
...     "weights": weights,
...     "altitude (m)": altitudes,
...     "temperature (K)": temperatures,
...     "scattering (m-1)": scatt,
...     "absorption (m-1)": absor,
...     "asymmetry": asymm,
...     "wavelength": wavelengths,
...     "band low": bandsLow,
...     "band up": bandsUp
... })
Assigning data to nodes ...
```

### makeAtmosphereFrom2D(data)

Generate a spherical atmosphere from single column data. The data are provided at the levels, i.e. at the interface between layers.

#### Parameters

**data (dict)** – Dictionnary with the following items:

- "nLevel"**  
[int] Number of levels.
- "nLat"**  
[int] Number of latitudes.
- "nCoeff": int**  
Number of quadrature points for the k-coeff.
- "nWavelengths": int**  
Number of wavelengths.
- "nAngle": int, optional**  
Number of angles in the phase functions (not required if using the builtin Henyey-Greenstein phase function).
- "weights": numpy.ndarray**  
The weights of the nCoeff quadrature points (shape=(nWavelengths, nCoeff)).
- "altitude (m)": numpy.ndarray**  
Array of altitudes (shape=(nLevel, nLat), [m]).
- "latitude (°)"**  
[numpy.ndarray] List of latitudes (shape=(nLat), [°]).
- "temperature (K)": numpy.ndarray**  
Array of temperatures (shape=(nLevel, nLat), [K]).
- "scattering (m-1)": numpy.ndarray**  
Array of scattering coefficients (shape=(nWavelengths, nLevel, nLat, nCoeff), [m-1]).
- "absorption (m-1)": numpy.ndarray**  
Array of absorption coefficients (shape=(nWavelengths, nLevel, nLat, nCoeff), [m-1]).
- "angles (°)": numpy.ndarray, optional**  
Array of angle values for the phase function (shape=(nAngle), float [°]). Only required when providing the phaseFunc.
- "phaseFunc": numpy.ndarray, optional**  
Array of discrete phase functions (shape = (nWavelengths, nLevel, nLat, nAngle, nCoeff)). Alternatively, the user can provide an array of asymmetry parameter.
- "asymmetry": numpy.ndarray, optional**  
Array of asymmetry parameter (shape=(nWavelengths, nLevel, nLat, nCoeff)). Alternatively, the user can provide the discrete phase function.
- "wavelength": numpy.ndarray, optional**  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the band center. The value is used of the phase function.
- "band low": numpy.ndarray**  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.
- "band up": numpy.ndarray**  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

## Notes

The nPhi parameter provided at the initialisation of the instance is used to define the longitudinal resolution of the atmospheric mesh and is therefore manatory. The nTheta parameter is obtained from the length of the latitude provided.

## Examples

```
>>> d = htrdrPy.Data(radius=1e6, nPhi=50, name="Planet")
>>> nLevel = 50
>>> nLat = 30
>>> nCoeff = 4
>>> nWavelengths = 20
>>> weights = np.array(nWavelengths * [0.2, 0.3, 0.3, 0.2]).
↳ reshape(nWavelengths, nCoeff)
>>> altitudes = np.tile(np.linspace(0, 5e5, nLevel), (nLat, 1)).T
>>> latitudes = np.linspace(-90, 90, nLat)
>>> temperatures = np.linspace(300, 500, nLevel*nLat).reshape(nLevel, nLat)
>>> scatt = np.linspace(1e-8, 1e-2,
...     nLevel*nLat*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...     nLat, nCoeff))
>>> absor = np.linspace(1e-5, 1e-1,
...     nLevel*nLat*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...     nLat, nCoeff))
>>> asymm = np.linspace(0, 1,
...     nLevel*nLat*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...     nLat, nCoeff))
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:] = bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> d.makeAtmosphereFrom2D({
...     "nLevel": nLevel,
...     "nLat": nLat,
...     "nCoeff": nCoeff,
...     "nWavelengths": nWavelengths,
...     "weights": weights,
...     "altitude (m)": altitudes,
...     "latitude (°)": latitudes,
...     "temperature (K)": temperatures,
...     "scattering (m-1)": scatt,
...     "absorption (m-1)": absor,
...     "asymmetry": asymm,
...     "wavelength": wavelengths,
...     "band low": bandsLow,
...     "band up": bandsUp
... })
```

### makeAtmosphereFrom3D(*data*)

Generate a spherical atmosphere from single column data. The data are provided at the levels, i.e. at the interface between layers.

#### Parameters

**data** (*dict*) – Dictionnary with the following items:

- **"nLevel"**  
[int] Number of levels.
- **"nLat"**  
[int] Number of latitudes.
- **"nLon"**  
[int] Number of longitudes.
- **"nCoeff": int**



Number of quadrature points for the k-coeff.

- **“nWavelengths”**: `int`  
Number of wavelengths.
- **“nAngle”**: `int, optional`  
Number of angles in the phase functions (not required if using the builtin Henyey-Greenstein phase function).
- **“weights”**: `numpy.ndarray`  
The weights of the nCoeff quadrature points (shape=(nWavelengths, nCoeff)).
- **“altitude (m)”**: `numpy.ndarray`  
Array of altitudes (shape=(nLevel, nLat, nLon), [m]).
- **“latitude (°)”**  
[`numpy.ndarray`] List of latitudes (shape=(nLat), [°]).
- **“longitude (°)”**  
[`numpy.ndarray`] List of longitudes (shape=(nLon), [°]).
- **“temperature (K)”**: `numpy.ndarray`  
Array of temperatures (shape=(nLevel, nLat, nLon), [K]).
- **“scattering (m-1)”**: `numpy.ndarray`  
Array of scattering coefficients (shape=(nWavelengths, nLevel, nLat, nLon, nCoeff), [m-1]).
- **“absorption (m-1)”**: `numpy.ndarray`  
Array of absorption coefficients (shape=(nWavelengths, nLevel, nLat, nLon, nCoeff), [m-1]).
- **“angles (°)”**: `numpy.ndarray, optional`  
Array of angle values for the phase function (shape=(nAngle), float [°]). Only required when providing the phaseFunc.
- **“phaseFunc”**: `numpy.ndarray, optional`  
Array of discrete phase functions (shape = (nWavelengths, nLevel, nLat, nLon, nAngle, nCoeff)). Alternatively, the user can provide an array of asymmetry parameter.
- **“asymmetry”**: `numpy.ndarray, optional`  
Array of asymmetry parameter (shape=(nWavelengths, nLevel, nLat, nLon, nCoeff)). Alternatively, the user can provide the discrete phase function.
- **“wavelength”**: `numpy.ndarray, optional`  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the band center. The value is used of the phase function.
- **“band low”**: `numpy.ndarray`  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.
- **“band up”**: `numpy.ndarray`  
Array of wavelength (shape = (nWavelengths), [m]). The values corresponds to the lower boundary of the band.

## Notes

The nTheta and nPhi parameters provided at the initialisation of the instance are not used and instead are defined from the length of the `latitude (°)` and `longitude (°)`, respectively.

## Examples

```

>>> d = htrdrPy.Data(radius=1e6, name="Planet")
>>> nLevel = 50
>>> nLat = 30
>>> nLon = 50
>>> nCoeff = 4
>>> nWavelengths = 20
>>> weights = np.array(nWavelengths * [0.2, 0.3, 0.3, 0.2]).
↳ reshape(nWavelengths, nCoeff)
>>> altitudes = np.moveaxis(np.tile(np.linspace(0, 5e5, nLevel),
...                                  (nLat, nLon, 1)), -1, 0)
>>> latitudes = np.linspace(-90, 90, nLat)
>>> longitudes = np.linspace(-180, 180, nLon)
>>> temperatures = np.linspace(300, 500, nLevel*nLat*nLon).reshape(nLevel,
↳ nLat, nLon)
>>> scatt = np.linspace(1e-8, 1e-2,
...                      nLevel*nLat*nLon*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...                                                                           nLat, nLon, nCoeff))
>>> absor = np.linspace(1e-5, 1e-1,
...                      nLevel*nLat*nLon*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...                                                                           nLat, nLon, nCoeff))
>>> asymm = np.linspace(0, 1,
...                      nLevel*nLat*nLon*nCoeff*nWavelengths).reshape((nWavelengths, nLevel,
...                                                                           nLat, nLon, nCoeff))
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:] = bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> d.makeAtmosphereFrom3D({
...     "nLevel": nLevel,
...     "nLat": nLat,
...     "nLon": nLon,
...     "nCoeff": nCoeff,
...     "nWavelengths": nWavelengths,
...     "weights": weights,
...     "altitude (m)": altitudes,
...     "latitude (°)": latitudes,
...     "longitude (°)": longitudes,
...     "temperature (K)": temperatures,
...     "scattering (m-1)": scatt,
...     "absorption (m-1)": absor,
...     "asymmetry": asymm,
...     "wavelength": wavelengths,
...     "band low": bandsLow,
...     "band up": bandsUp
... })

```

**makeFromLMDZ**(*LMDZinput*, *LMDZouput*, *weights=None*, *keys*={'altitude': 'altitude', 'assym': 'gv',  
'extinction': 'kv', 'geopotential': 'pphi', 'geopotentialSurf': 'pphis', 'latitude': 'lat',  
'longitude': 'lon', 'pressure': 'p', 'ssa': 'wv', 'temp': 'temp', 'tsurf': 'tsurf', 'wavelength':  
'wavelength\_vi'}, *wavelength=None*, *time=-1*, *hg=False*, *nAngle=181*,  
*phaseFunc*=<function Data.<lambda>>)

Generate an heterogeneous sphere from LMDZ output and input files.

## Parameters

- **LMDZinput** (*str*) – Path to the netCDF file containing surface information, to be read.
- **LMDZouput** (*str*) – Path to the netCDF file containing atmosphere information, to be read.
- **weights** (*numpy.ndarray*, optional) – Array containing the weights of the Gaussian quadrature points for correlated-k data (*shape*=(*nWeight*)). If not provided, assumes no correlated-k are used. It is not required if wavelengths are separated (see *wavelengths*).
- **keys** (*{str : str}*, *default htrdrPy.keysLMDZtitan\_vi*) – Dictionnary of correspondance between keys used in this method and keys from the input/output file. It must contain the following keys:
  - **'tsurf'**  
[default 'tsurf'] Key for the surface temperature.
  - **'temp'**  
[default 'temp'] Key for the atmospheric temperature.
  - **'wavelength'**  
[default 'wavelength\_vi'] Key for the wavelengths.
  - **'ssa': default 'wv'**  
Key for the single scattering albedo.
  - **'extinction'**  
[default 'kv'] Key for the extinction coefficient.
  - **'assym'**  
[default 'gv'] Key for the asymmetry parameter.
  - **'geopotential'**  
[default 'pphi'] Key for the geopotential.
  - **'altitude'**  
[default 'altitude'] Key for the altitude.
  - **'latitude'**  
[default 'lat'] Key for the latitude.
  - **'longitude'**  
[default 'lon'] Key for the longitude.
  - **'pressure'**  
[default 'p'] Key for the pressure.
- **wavelengths** (*{str : dict}*, *optional*) – Dictionnary containing the wavelength bands data. It must be composed of 1 sub-dictionnary for every band.
  - **'index'**  
[dict] Dictionnary containing the data for a specific band ('index' is the index referring to the band in the GCM output file, e.g. 'v\_23' for the last visible band in Titan GCM). The sub-dictionnary must contain the following items:
    - **'wavelength'**  
[float] Central wavelength of the band (in m).
    - **'low'**  
[float] Lower bound of the band (in m).
    - **'up'**  
[float] Upper bound of the band (in m).
    - **'weights'**  
[*numpy.ndarray*, optional] The weights associated to the different k-correlated coefficients (*shape*=(*nWeights*)). If no k-coeff are used, omit the key.
- **time** (*int*, *default -1*) – Time index to use, default is last.

- **hg** (*bool, default False*) – Whether or not to use the Henyey-Greenstein phase function built in htldr
- **nAngle** (*int, optional, default 181*) – Number of angles to use in the discrete phase functions. (If hg=True nAngle is omitted).
- **phaseFunc** (*func, optional, default = 1 + g cos(theta)*) – Function to calculate the discrete phase function. The function must take in first argument the asymetry parameter and in second argument the angle (in rad). If hg=True phaseFunc is omitted.

## Notes

If the surface temperature is not provided in the output file, it will be read from the input file.

**makeGroundFrom1D**(*surfaceTemperature, brdf*)

Generates a spherical ground considering uniform temperature and optical properties.

### Parameters

- **surfaceTemperature** (*float*) – Temperature of the surface [K].
- **brdf** (*dict*) – Surface reflexion properties with the following items:
  - **“kind”**  
[[“lambertian”, “specular”]] Kind of brdf function to use.
  - **“albedo”**  
[*numpy.ndarray*] Wavelength dependent surface albedos  
(*shape*=(*nWavelength*)).
  - **“wavelengths”**  
[*numpy.ndarray, optional*] Wavelengths where the albedo is defined  
(*shape*=(*nWavelength*), [m]). Alternatively, the user can specify the bands with the “bands” keyword.
  - **“bands”**  
[*numpy.ndarray, optional*] Wavelengths bands where the albedo is defined  
(*shape*=(*nWavelength*,2), [m]). The values corresponds to the bands limits.

## Notes

The nTheta and nPhi parameters provided at the initialisation of the instance are used to define the resolution of the ground mesh and are therefore manatory.

## Examples

```
>>> d = htldrPy.Data(radius=1e6, nTheta=30, nPhi=50, name="Planet")
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:], bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> surfTemp= 300
>>> surfAlb= np.ones(nWavelengths) * 0.5
>>> d.makeGroundFrom1D(surfTemp, {
...     "kind": "lambertian",
...     "albedo": surfAlb,
...     "bands": np.array([bandsLow, bandsUp]).T
... })
Mesh generator. Ntheta = 30, Nphi = 50, R = 1000000.0
Generating points...
```

(continues on next page)

(continued from previous page)

```

Generating needs...
triangles,rectangles generation completed. N_triangles = 98, N_rectangles = 1323
Generating triangles...
generating bin...

```

**makeGroundFrom1D\_PP**(*surfaceTemperature*, *brdf*)

Generates a plan-parallel ground considering uniform temperature and optical properties.

**Parameters**

- **surfaceTemperature** (*float*) – Temperature of the surface [K].
- **brdf** (*dict*) – Surface reflexion properties with the following items:
  - **"kind"**  
[["lambertian", "specular"]] Kind of brdf function to use.
  - **"albedo"**  
[*numpy.ndarray*] Wavelength dependent surface albedos  
(shape=(nWavelength)).
  - **"wavelengths"**  
[*numpy.ndarray*, optional] Wavelengths where the albedo is defined  
(shape=(nWavelength), [m]). Alternatively, the user can specify the bands  
with the "bands" keyword.
  - **"bands"**  
[*numpy.ndarray*, optional] Wavelengths bands where the albedo is defined  
(shape=(nWavelength,2), [m]). The values corresponds to the bands limits.

**Notes**

In plan-parallel mode, the *radius* parameter passed at the initialisation of the instance is used as the horizontal expansion of the ground. Make sure to use a large enough value. Also, the *nTheta* and *nPhi* parameters are not used.

**Examples**

```

>>> d = htrdrPy.Data(radius=1e6, name="Planet")
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:], bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> surfTemp= 300
>>> surfAlb= np.ones(nWavelengths) * 0.5
>>> d.makeGroundFrom1D_PP(surfTemp, {
...     "kind": "lambertian",
...     "albedo": surfAlb,
...     "bands": np.array([bandsLow, bandsUp]).T
... })
generating bin...

```

**makeGroundFrom2D**(*surfaceTemperature*, *brdf*)

Generates a spherical ground considering temperature and optical properties varying along the latitude.

**Parameters**

- **surfaceTemperature** (`numpy.ndarray`) – Temperature of the surface (`shape=(nLat)`, [K]).
- **brdf** (`dict`) – Surface reflexion properties with the following items:
  - **“kind”**  
[["lambertian", "specular"]] Kind of brdf function to use.
  - **“albedo”**  
[`numpy.ndarray`] Wavelength dependent surface albedos (`shape=(nWavelength, nLat)`).
  - **“latitude”**  
[`numpy.ndarray`] List of latitudes (`shape=(nLat)`, [°]).
  - **“wavelengths”**  
[`numpy.ndarray`, optional] Wavelengths where the albedo is defined (`shape=(nWavelength)`, [m]). Alternatively, the user can specify the bands with the “bands” keyword.
  - **“bands”**  
[`numpy.ndarray`, optional] Wavelengths bands where the albedo is defined (`shape=(nWavelength, 2)`, [m]). The values corresponds to the bands limits.

## Notes

The `nPhi` parameter provided at the initialisation of the instance is used to define the longitudinal resolution of the ground mesh and is therefore manatory. The `nTheta` parameter is obtained from the length of the `latitude` provided.

## Examples

```
>>> d = htrdrPy.Data(radius=1e6, nPhi=50, name="Planet")
>>> nLat = 30
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:], bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> latitudes = np.linspace(-90, 90, nLat)
>>> surfTemp = 300 * np.ones_like(latitudes)
>>> surfAlb = np.ones((nWavelengths, nLat)) * 0.5
>>> d.makeGroundFrom2D(surfTemp, {
...     "kind": "lambertian",
...     "albedo": surfAlb,
...     "latitude": latitudes,
...     "bands": np.array([bandsLow, bandsUp]).T
... })
Mesh generator. Ntheta = 30, Nphi = 50, R = 1000000.0
Generating points...
Generating noeds...
triangles,rectangles generation completed. N_triangles = 98, N_rectangles = 1323
Generating triangles...
generating bin...
```

### **makeGroundFrom3D**(*SurfaceTemperature*, *brdf*)

Generates a spherical ground considering temperature and optical properties varying along the latitude and longitude.

#### Parameters

- **surfaceTemperature** (`numpy.ndarray`) – Temperature of the surface (shape=(`nLat`, `nLon`), [K]).
- **brdf** (`dict`) – Surface reflexion properties with the following items:
  - **“kind”**  
[["lambertian", "specular"]] Kind of brdf function to use.
  - **“albedo”**  
[`numpy.ndarray`] Wavelength dependent surface albedos (shape=(`nWavelength`, `nLat`, `nLon`)).
  - **“latitude”**  
[`numpy.ndarray`] List of latitudes (shape=(`nLat`), [°]).
  - **“longitude”**  
[`numpy.ndarray`] List of longitudes (shape=(`nLon`), [°]).
  - **“wavelengths”**  
[`numpy.ndarray`, optional] Wavelengths where the albedo is defined (shape=(`nWavelength`), [m]). Alternatively, the user can specify the bands with the “bands” keyword.
  - **“bands”**  
[`numpy.ndarray`, optional] Wavelengths bands where the albedo is defined (shape=(`nWavelength`, 2), [m]). The values corresponds to the bands limits.

## Notes

The `nTheta` and `nPhi` parameters provided at the initialisation of the instance are not used and instead are defined from the length of the `latitude` and `longitude`, respectively.

## Examples

```
>>> d = htrdrPy.Data(radius=1e6, name="Planet")
>>> nLat = 30
>>> nLon = 50
>>> wavelengths = np.linspace(2e7, 9e7, nWavelengths)
>>> bandsLow = np.zeros(nWavelengths)
>>> bandsUp = np.zeros(nWavelengths)
>>> bandsLow[1:], bandsUp[:-1] = 0.5 * (wavelengths[1:] + wavelengths[:-1])
>>> bandsLow[0] = 1.5 * wavelengths[0] - 0.5 * wavelengths[1]
>>> bandsUp[-1] = 1.5 * wavelengths[-1] - 0.5 * wavelengths[-2]
>>> latitudes = np.linspace(-90, 90, nLat)
>>> longitudes = np.linspace(-180, 180, nLon)
>>> surfTemp = 300 * np.ones((nLat, nLon))
>>> surfAlb= np.ones((nWavelengths, nLat, nLon)) * 0.5
>>> d.makeGroundFrom3D(surfTemp, {
...     "kind": "lambertian",
...     "albedo": surfAlb,
...     "latitude": latitudes,
...     "longitude": longitudes,
...     "bands": np.array([bandsLow, bandsUp]).T
... })
Mesh generator. Ntheta = 30, Nphi = 50, R = 1000000.0
Generating points...
Generating noeds...
triangles,rectangles generation completed. N_triangles = 98, N_rectangles = 1323
Generating triangles...
generating bin...
```

**writeInputAtmosphere()**

Write the atmosphere binary input files for htrdr-planets

**writeInputGround()**

Write the ground binary input files for htrdr-planets

**writeInputs**(*octree\_def=512, opthick=1, nthOctree=8, procOctree='master', octreeFile=""*)

Write the binary input files for htrdr-planets and precalculate octrees if the *octreeFile* is passed.

**Parameters**

- **octree\_def** (*int or str, default 512*) – Maximal definition of the octree grid.
- **opthick** (*float or str, default 1*) – Optical thickness threshold to assess the merge of cells.
- **nthOctree** (*int, default 8*) – Number of threads to use for octree computation.
- **procOctree** (*{'all', 'master'}, default 'master'*) – Which process must realize the octree calculation (useless if no storage). Put to 'all' if the processes do not share the disk space.
- **octreeFile** (*str, optional*) – Filename to use for the storing of octrees. If not provided, octrees are not stored on disk. If a file is provided, a “blank run” of htrdr-planets is launched to precalculate octrees. The file is located in the *outputs\_{name}* folder.

**Examples**

```
>>> d = htrdrPy.Data(radius=1e6, nPhi=50, name="Planet")
...
>>> d.writeInputs(octreeFile="octree.bin")
```

This will generate the input files and start a very quick run of htrdr-planets to generate an octree file that will be stored in *outputs\_Planet/octree.bin*

```
>>> d.writeInputs(octree_def=1024)
```

This will generate the input files and keep the information on the *octree\_def* for the future calculation of the octrees. The octrees are not precalculated since they will only be stored in memory.

**writeVTKfiles()**

Write the VTK and obj files for view in paraview or other visualisation software handling obj and VTK files.

**writeVTKfilesAtmosphere()**

Write the atmosphere VTK and obj files.

**writeVTKfilesGround()**

Write the ground VTK and obj files.

### 3.2.2 htrdrPy.geometry module

```
class htrdrPy.geometry.Geometry(source=None, camera=None, image=None, volrad=None,
                                case=None)
```

Bases: object

*htrdrPy.Geometry* is a class that aims at managing all information related to the observation geometry. It handles the camera, source and image properties as well as the mesh on which running the volumic radiative budget calculations, if running on this mode.

The Geometry module handles the positioning, orientation and properties of the camera and source. The data can be provided directly when creating an instance as distinct dictionaries for the source, the camera and the image, with the following keys:



## Parameters

- **source** (*dict*, *optional*) – Dictionary containing the information on the source, with the following keys:
  - **"longitude"**  
[float] Longitude of the source [°].
  - **"latitude"**  
[float] Latitude of the source [°].
  - **"distance"** :  
Distance of the source [m].
  - **"radius"**  
[float] Radius of the source [m].
  - **"temperature"**  
[float, optional] Surface temperature of the source [K] (used to calculate the Planck's function)
  - **"radiance"**  
[str, optional] Path to a radiance file in htdr readable format.
  - **"spectrum"**  
[numpy.ndarray, optional] 2-D array containing the spectrum (shape=(nWavelength,2)). The first column contains the wavelength [nm] and the second contains the radiance [W/m2/sr/nm] at the surface of the source.
- **camera** (*dict*, *optional*) – Dictionary containing the information on the camera, with the following keys:
  - **"position"**  
[numpy.ndarray] Position of the camera in cartesian coordinates (shape=(3), [m]). The origin corresponds to the center of the observed target.
  - **"target"**  
[numpy.ndarray] Position of the target in cartesian coordinates (shape=(3), [m]). The origin corresponds to the center of the observed target. This is NOT the line of sight but the position vector of the target.
  - **"field of view"**  
[float] Vertical field of view of the camera [°] (the horizontal field of view is calculated via scaling by the image aspect ratio, assuming square pixels).
  - **"roll"**  
[numpy.ndarray, optional] Vector setting the upward direction of the camera (shape=(3), [m]) i.e. a vector in the pixel plane to turn the camera around the line of sight. If not provided, it is calculated as perpendicular to the line of sight and to z axis (or x axis if the z axis corresponds to the line of sight).
- **image** (*dict*, *optional*) – Dictionary containing the information on the image, with the following keys:
  - **"definition"**  
[array-like] Definition (number of pixels) of the image (shape=(2), the first value is the horizontal number of pixel and the second value is the vertical pixel count).
  - **"sampling"**  
[int] Number of rays to sample for each pixel.

## exportGeometry()

Export the geometry (source, camera and image data) in a geometry\_{case}.json file stored in the "geometries/" repository

**geometryFromAPIE**(*observation, distance, radius, cameraFOV, sourceDist, sourceRad, sourceTemp=None, radianceFile=None, spectrum=None*)

Calculate the observation geometry from Azimut, Phase, Incident and Emergent angles.

#### Parameters

- **observation** (*dict*) – Dictionnary containing the geometry of the observation with the following items:
  - **“azimut”**  
[float] Angle between the projected incidence and the projected emergence [°].
  - **“phase”**  
[float] Angle between incident rays (directly from the source) and the line of sight [°].
  - **“incidence”**  
[float] Angle between the incident rays (directly from the source) and the normal to the ground at the observed loaction [°].
  - **“emergence”**  
[float] Angle between the normal to the line of sight and the normal to the ground at the observed loaction [°].
- **distance** (*float*) – Distance from the camera to the target [m].
- **radius** (*float*) – Radius of the planet to locate the target on the surface [m].
- **cameraFOV** (*float*) – Field of view of the camera (c.f. `htrdrPy.Geometry.setCamera`).
- **sourceDist** (*float*) – Distance of the source [m] (c.f. `htrdrPy.Geometry.setSource`).
- **sourceRad** (*float*) – Radius of the source [m] (c.f. `htrdrPy.Geometry.setSource`).
- **spectrum** (*float, optional*) – Spectrum of the source (c.f. `htrdrpy.geometry.setsource`).
- **radianceFile** (*float, optional*) – Path to the radiance file of the source (c.f. `htrdrpy.geometry.setsource`).
- **sourceTemp** (*float, optional*) – Temperature of the source (c.f. `htrdrpy.geometry.setsource`).

**plotGeometry**(*ax, radius*)

Plot the gometry: the observed planet, the line of sight (blue vector), the source direction (red vector), the camera plan (black vectors) and the field of view (green vectors).

#### Parameters

- **ax** (*matplotlib.pyplot.Axes*) – Axe on which draxing the plot. It must be a 3D axe.
- **radius** (*float*) – Radius of the planet [m].

#### Warning

matplotlib.pyplot 3d projection has some issues with vector orientation. If they have the right direction, they may not have the right sens.

**setCamera**(*position, targetPosition, fieldOfView, roll=None*)

Setup the camera position, orientation and field of view

#### Parameters

- **position** (`numpy.ndarray`) – Position of the camera in cartesian coordinates (shape=(3), [m]). The origin corresponds to the center of the observed target.
- **targetPosition** (`numpy.ndarray`) – Position of the target in cartesian coordinates (shape=(3), [m]). The origin corresponds to the center of the observed target. This is NOT the line of sight but the position vector of the target.
- **fieldOfView** (`float`) – Vertical field of view of the camera [°] (the horizontal field of view is calculated via scaling by the image aspect ratio, assuming square pixels).
- **roll** (`numpy.ndarray`, optional) – Vector setting the upward direction of the camera (shape=(3), [m]) i.e. a vector in the pixel plane to turn the camera around the line of sight. If not provided, it is calculated as perpendicular to the line of sight and to z axis (or x axis if the z axis corresponds to the line of sight).

**setImage**(*definition, sampling*)

Setup the image properties

#### Parameters

- **definition** (*array-like*) – Definition (number of pixels) of the image (shape=(2), the first value is the horizontal number of pixel and the second value is the vertical pixel count).
- **sampling** (*int*) – Number of rays to sample for each pixel.

**setSource**(*longitude, latitude, distance, radius, temperature=None, radianceFile=None, spectrum=None*)

Setup the source properties.

#### Parameters

- **longitude** (`float`) – Longitude of the source [°].
- **latitude** (`float`) – Latitude of the source [°].
- **distance** – Distance of the source [m].
- **radius** (`float`) – Radius of the source [m].
- **temperature** (`float`, optional) – Surface temperature of the source [K] (used to calculate the Planck's function)
- **radianceFile** (*str*, optional) – Path to a radiance file in htrdr readable format.
- **spectrum** (`numpy.ndarray`, optional) – 2-D array containing the spectrum (shape=(nWavelength,2)). The first column contains the wavelength [nm] and the second contains the radiance [W/m<sup>2</sup>/sr/nm] at the surface of the source.

#### Notes

Whereas `temperature`, `radianceFile` and `spectrum` are optional, at least one of those must be provided. If more than one is provided, the `spectrum` is taken in priority, then the `radianceFile` and finally the `temperature`.

**setSpectralCumulDist**(*pdf, data=None*)

Setup the probability density function to used for sampling the wavelength and correlated-k coefficient.

#### Parameters

**pdf** (`numpy.ndarray`) – Table containing the cumulative distribution used to sample the wavelength and k-coefficient (shape=(nAltitudes,nLatitudes,nLongitudes,nSpectralElements)). nSpectralElements correspond to nWavelength \* nCoeff.

**setVolrad**(*sampling*, *mesh*='origin', *args*=None)

Setup the volumic radiative budget properties. The volumic radiative budget mode evaluate the divergence of the flux within each provided tetrahedron. This method handles the generation of the mesh on which the calculation will be conducted.

#### Parameters

- **sampling** (*int*) – Number of ray to sample for each tetrahedron.
- **mesh** (*str*) – Method to build the mesh on which the radiative budget calculation is realized. The default value is "origin". The possible values are: "origin", "makeColumnPP", "makeFromCellCoord", "makeSliceAltLat", "extractFromData".

Table 1: Building method

"origin"	Use the original atmosphere mesh.
"makeColumnPP"	Generate one plane-parallel column.
"makeFromCellCoord"	Generate a complete sphere from a table of coordinates.
"makeSliceAltLat"	Generate a slice in altitude / latitude from a table of coordinates.
"extractFromData"	Extract a list of cells from an already generated mesh.

- **args** (*tuple*) – Tuple containing the arguments required by the chosen mesh generation method.

Table 2: Arguments

Method	Required arguments
"origin"	
"makeColumnPP"	<p><b>altitudes</b> [numpy.ndarray] Array containing the altitudes [m], either at the center of the cells or at the boundaries (shape = nLevel or nLayer).</p> <p><b>hwidth</b> [float] Horizontal dimension of the squared base column [m].</p> <p><b>center</b> [bool, default: True] True if the altitudes represent the cell centers and False if they correspond to the boundaries of the cells.</p>
"makeFromCellCoord"	<p><b>cellCoord</b> [numpy.ndarray] Coordinates of the cells centers or cell interface centers (c.f. <code>onLevels</code>, shape=(nAltitudes,nLatitudes,nLongitudes,3), [m])</p> <p><b>radius</b> [float] Radius of the planet [m].</p> <p><b>poles</b> [bool, default False] Whether or not the first and last latitudes correspond to the poles in the given array of coordinates.</p> <p><b>onLevels</b> [bool, default False] Whether or not the altitudes provided in the coordinates are given on the levels or in the center of the cells.</p>
"makeSliceAltLat"	<p><b>cellCoord</b> [numpy.ndarray] Coordinates of the cells centers or cell interface centers (c.f. <code>onLevels</code>, shape=(nAltitudes,nLatitudes,3), [m])</p> <p><b>radius</b> [float] Radius of the planet [m].</p> <p><b>dLongitude</b> [float] Longitude width of the slice [°].</p> <p><b>poles</b> [bool, default False] Whether or not the first and last latitudes correspond to the poles in the given array of coordinates.</p> <p><b>onLevels</b> [bool, default False] Whether or not the altitudes provided in the coordinates are given on the levels or in the center of the cells.</p>
"extractFromData"	<p><b>data</b> [htrdrPy.Data or htrdrPy.Geometry] Object instance containing an already generated mesh from which extracting the list of cells. This is non-destructive for the original <code>htrdrPy.Data</code> or <code>htrdrPy.Geometry</code> and only affects the current instance that "steals" the wanted cells.</p>

### 3.2.3 htrdrPy.script module

**class** htrdrPy.script.**Script**(*case=""*, *threadFlag=""*, *MPIcmd=""*, *verbose=True*)

Bases: object

The htrdrPy.Script module aims at creating a callable (a function) taking as input a htrdrPy.Data object.

#### Examples

The first step is to create an instance of htrdrPy.Script:

```
>>> script = htrdrPy.script(case="caseName")
```

The second step is to define the kind of script to be executed (see the documentation for all possibilities). If none of the predefined script suits your use, you can use the htrdrPy.Script.startMultipleObsGeometry method.

```
>>> script.startMultipleObsGeometry(...)
```

Finally, you can call the script on an already defined htrdrPy.Data object:

```
>>> script(data)
```

The Script module aims at creating a callable (a function) to be called on a Data object.

#### Parameters

- **case** (*str*, *optional*) – String to identify the script. This is mostly usefull in case different scripts are used on the same htrdrPy.Data instance. The output files being stored in the same folder (see htrdrPy.Data documentation), the case name is used to differentiate the output files of the different scripts.
- **threadFlag** (*str*, *optional*) – Thread option to use in the htrdr command. Should take the form “-t <num>” where <num> is the number of threads to be used. If left empty, the maximum number of threads (corresponding to the number of virtual cores on the computer) will be used.
- **MPIcmd** (*str*, *optional*) – MPI command to pass before the call to htrdr-planets, depending on the MPI runner on your system. For insatnce, it could be an ‘mpirun’ command on many computers, or a ‘srun’ command on a slurm-based supercalculator.
- **verbose** (*bool*, *default True*) – Whether or not activate the verbose of htrdr-planets.

**bandIntegratedImage**(*geometry: Geometry*, *kind*, *wavelengthLow*, *wavelengthUp*)

Set up the script to calculate an image integrated over a given spectral range.

#### Parameters

- **geometry** (htrdrPy.Geometry) – A htrdrPy.Geometry object previously created and set up.
- **kind** ({*"sw"*, *"lw"*}) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelengthLow** (*float*) – Lower boundary wavelength [m].
- **wavelengthUp** (*float*) – Upper boundary wavelength [m].

**compositeRBG**(*geometry: Geometry*, *kind*, *wavelengthRed*, *wavelengthGreen*, *wavelengthBlue*)

Set up the script to calculate the composite image from 3 monochromatic images.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelengthRed** (`float`) – Wavelength for red channel image [m].
- **wavelengthGreen** (`float`) – Wavelength for green channel image [m].
- **wavelengthBlue** (`float`) – Wavelength for red channel image [m].

#### Warning

This script has not been tested yet and may not work correctly. This should be treated in future versions.

**imageRatio**(*geometry*: `Geometry`, *kind*, *wavelengthNum*, *wavelengthDen*)

Set up the script to calculate the ratio of 2 monochromatic images.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelengthNum** (`float`) – Wavelength for numerator image [m].
- **wavelengthDen** (`float`) – Wavelength for denominator image [m].

**monochromaticImage**(*geometry*: `Geometry`, *kind*, *wavelength*)

Set up the script to calculate a monochromatic image.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelength** (`float`) – Wavelength to use for the calculation [m].

**reflectanceSpectrum**(*geometry*: `Geometry`, *kind*, *wavelengths*, *bandWidths*=`None`)

Start multiple runs of htrdr-planets to calculate a reflectance spectrum, producing an output for each wavelength.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelength** (`float`) – Wavelength to use for the calculation [m].
- **bandWidths** (`numpy.ndarray`) – Integration band around each wavelength. If not provided, the calculation is monochromatic (`shape=(nWavelengths), [m]`).

**spectrum**(*geometry*: `Geometry`, *kind*, *wavelengths*, *bandWidths*=`None`)

Start multiple runs of htrdr-planets to calculate a spectrum, producing an output for each wavelength.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.
- **wavelength** (`float`) – Wavelength to use for the calculation [m].
- **bandWidths** (`numpy.ndarray`) – Integration band around each wavelength. If not provided, the calculation is monochromatic (`shape=(nWavelength)`, [m]).

**startMultipleObsGeometry**(*obsList*: `list[Geometry]`, *wavelength*)

Start multiple runs with different observation geometries but the same planet inputs

#### Parameters

- **obsList** (*array-like*) – List of `htrdrPy.Geometry` instances.
- **wavelength** (*dict*) – Dictionary containing the spectral information with the following items:
  - **“type”**  
[`“cie_xyz”, “sw”, “lw”`]] Type of calculation.
  - **“low”**  
[`float`] Lower bound of integration band [m].
  - **“up”**  
[`float`] Upper bound of integration band [m] (if monochromatic calculation, “up” = “low”).

**startRadBudgetGCM**(*geometry*: `Geometry`, *kind*)

Set up the script to calculate the radiative budget of each GCM cell.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.
- **kind** (`{"sw", "lw"}`) – Type of calculation. “sw” for a calculation with an external source, and “lw” to use the atmosphere emission as source.

**visibleImage**(*geometry*: `Geometry`)

Set up the script to calculate a visible (RGB) image.

#### Parameters

- **geometry** (`htrdrPy.Geometry`) – A `htrdrPy.Geometry` object previously created and set up.

`htrdrPy.script.loadScript`(*filename*)

Load a `htrdrPy.Script` object from the binary file where it is saved (generated after the call on a `htrdrPy.Data` object).

#### Parameters

- **filename** (*str*) – Path to the binary file to be loaded.

### 3.2.4 htrdrPy.postprocess module

**class** `htrdrPy.postprocess.Postprocess`(*script*: `Script` = `None`, *skip*=`False`, *localPath*=`False`, *kwargs*=`{}`)

Bases: `object`

The `htrdrPy.Postprocess` module aims at providing tools to process the outputs generated by `htrdr-planets` or auto-process the outputs based on the `htrdrPy.Script` used.



## Examples

In the case of the use of a predefined `htdrPy.Script` other than `htdrPy.Script.startMultipleObsGeometry`, the user simply needs to provide the `htdrPy.Script` used at the creation of the `htdrPy.Postprocess` instance.

```
>>> pp = htldrPy.Postprocess(script)
```

This will generate the output files (‘.json’, images, ...) in a ‘result\_{name}/’ folder, where the ‘name’ is the name of the `htdrPy.Data` instance passed to the `htdrPy.Script`.

### Parameters

- **script** (`htdrPy.Script`, optional) – `htdrPy.Script` object which has been called on a `htdrPy.Data` object.
- **skip** (*bool*, *default False*) – Whether or not to skip the default post-processing operation. Default is False.
- **localPath** (*bool*, *default False*) – Whether or not to create the result directory at the root of the script running the post-processing. If True, the result directory will be located in the same directory as the running post-processing script, otherwise, it is located in the directory where `htdr-planets` was run.
- **kwargs** (*dict*) – Dictionary containing the arguments for the post-processing routine. You will find the required items, depending on the script used, in the following table.

Table 3: kwargs

Script	Items
<ul style="list-style-type: none"> <li>• htrdPy. Script. visibleImage or</li> <li>• htrdPy. Script. monochromaticI or</li> <li>• htrdPy. Script. bandIntegrated</li> </ul>	<ul style="list-style-type: none"> <li>• <b>“exposure”</b> [float, default 1.] Exposure time [s] for the generation of the image.</li> <li>• <b>“cmap”</b> [str, default “inferno”] Color map to be used. See htrdr documentation for all possibilities.</li> </ul>
htrdPy.Script. imageRatio	<ul style="list-style-type: none"> <li>• <b>“threshold”</b> [float, default 0.] Threshold limit below which the data are considered zero (expressed relatively to the data maximale value: <math>\text{threshold} * \max(\text{data})</math>).</li> </ul>
<ul style="list-style-type: none"> <li>• htrdPy. Script. spectrum or</li> <li>• htrdPy. Script. reflectanceSpe or</li> <li>• htrdPy. Script. compositeRBG</li> </ul>	
htrdPy.Script. startRadBudgetG	<ul style="list-style-type: none"> <li>• <b>“heatCapacity”</b> [float, optional] Heat capacity of the atmosphere [J/kg/K]. If not provided, the calculation of the heating rate is not realised and the resulting file only contains the flux divergence.</li> <li>• <b>“rho”</b> [float, optional] Mass volume density of the atmosphere [kg/m<sup>3</sup>]. If not provided, the calculation of the heating rate is not realised and the resulting file only contains the flux divergence.</li> </ul>

## Notes

In most cases, the module recognize the kind of script and automatically apply the required post-process function. This is not the case for script based on “startMultipleObsGeometry”, as the post-processing routine to be used is not trivial. A bunch of additional methods are therefore provided.

**extractMeanRadianceFromOutput** (*file*, *time=False*)

Calculate the average radiance and standard deviation over all pixels

**Parameters**

**file** (*str*) – Path to the htdr output file containing the pixels information.

**Returns**

- *float* – Average radiance over the pixel grid (units are similar to htdr outputs).
- *float* – Standard deviation of the radiance over the pixel grid (units are similar to htdr outputs).
- *float* – Mean computation time per path [ $\mu$ s].
- *float* – Standard deviation of the computation time per path [ $\mu$ s].

**extractMeanRadiances**(*indices=False*)

Extract the mean radiances of all (or a subset) images generated by the Script.

**Parameters**

**indices** (*array-like*) – List containing the indices of the file to be processed. In case multiple outputs are generated (can be the case for a spectrum, or using `htdrPy.Script.startMultipleGeometry`) the file index follows the order of execution of the runs (for a spectrum, the order in which the wavelength have been given and for `htdrPy.Script.startMultipleGeometry` the order of the list of geometries.)

**Returns**

Dictionnary with mean radiances and standard deviations (c.f. `htdrPy.Postprocess.extractMeanRadianceFromOutput`) for all requested output files.

**Return type**

dict

**getImage**(*file*)

Recover the image data from a htdr output file.

**Parameters**

**file** (*str*) – Path to the htdr output file.

**Returns**

- `numpy.ndarray` – Radiances associated to each pixel (shape=(nPixelX,nPixelY)).
- `numpy.ndarray` – Standard deviation associated to each pixel (shape=(nPixelX,nPixelY)).

**getImages**()

Recover the images data from all htdr output files generated by the Script.

**Returns**

Dictionnary with radiances and standard deviations map for all output files.

**Return type**

dict

**processImages**(*exposure=1, cmap='inferno'*)

Generates the image corresponding to all output files.

**Parameters**

- **exposure** (*float, default 1.*) – Exposure time [s] for the generation of the image.
- **cmap** (*str, default "inferno"*) – Color map to be used. See htdr documentation for all possibilities.

**processSingleArrayObsSW**()

Processes the previously generated files in the case of single array (one of the image definition is 1).

**Returns**

Dictionnary containing 2-D `numpy.ndarray` (shape=(nPixel,2)) for each output file. The

first column of the arrays are radiances (units are similar to htrdr outputs) and the second columns are the associated standard deviation (units are similar to htrdr outputs).

**Return type**

dict

### 3.2.5 htrdrPy.helperFunctions module

`htrdrPy.helperFunctions.cart2sphere(vec)`

Convert cartesian to spherical coordinates.

**Parameters**

**vec** (`numpy.ndarray`) – Cartesian coordinate array (x [m], y [m], z [m]).

**Returns**

Spherical coordinate array (altitude [m], latitude [°], longitude [°]).

**Return type**

`numpy.ndarray`

`htrdrPy.helperFunctions.combineEstimates(sumX, sumXsquare, numbers)`

Calculate the mean, the variance and the standard deviation of a set of estimates.

**Parameters**

- **sumX** (`numpy.ndarray`) – Array containing the sum of the Monte Carlo weights of a list of estimates.
- **sumXsquare** (`numpy.ndarray`) – Array containing the sum of the square of the Monte Carlo weights of a list of estimates.
- **numbers** (`numpy.ndarray`) – Array containing the number of realizations for each estimate.

**Returns**

- *float* – Mean of the combined estimates.
- *float* – Variance of the combined estimates.
- *float* – Standard deviation of the combined estimates.

`htrdrPy.helperFunctions.dplanck_dT(T, wvl, r_d=False)`

Calculate the derivative of the planck emission regarding the temperature in W/m<sup>2</sup>/sr/m/K for a surface at temperature T and at wavelengths wvl.

**Parameters**

- **T** (*float* or `numpy.ndarray`) – Temperature or N-D array of temperatures [K] of the emitting surface.
- **wvl** (*float* or `numpy.ndarray`) – Wavelength or 1-D array of wavelengths.
- **(optional (r\_d))** – Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.
- **(shape=(2)) (1-D array)** – Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.
- **[m]) (float)** – Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.

**Returns**

Radiance either at the surface of the source (if `r_d` not provided) or at the given distance from the source. The shape depends on the shape of the parameters provided. If both the

T and wvl are floats, the result is a float. If T is a float and wvl an array, the result has the length of wvl. If T is an array and wvl is a float, the result has the shape of T. Finally, if both T and wvl are arrays (T has dimension N and wvl has dimension 1), the result has the N+1 dimensions (the N dimensions of T plus the dimension of wvl).

#### Return type

float or M-D array (shape=(nWavelength), float [W/m<sup>2</sup>/sr/m]))

htrdrPy.helperFunctions.**planck**(T, wvl, r\_d=False)

Calculate planck emission in W/m<sup>2</sup>/sr/m for a surface at temperature T and at wavelengths wvl.

#### Parameters

- **T** (float or `numpy.ndarray`) – Temperature or N-D array of temperatures [K] of the emitting surface.
- **wvl** (float or `numpy.ndarray`) – Wavelength or 1-D array of wavelengths.
- **(optional (r\_d))** (Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.
- **(shape=(2)) (1-D array)** – Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.
- **[m])** (float) – Shape 2 array containing the source radius and distance, respectively. If not provided, returns the surface radiance, if given, returns the radiance received at that distance from the source.

#### Returns

Radiance either at the surface of the source (if r\_d not provided) or at the given distance from the source. The shape depends on the shape of the parameters provided. If both the T and wvl are floats, the result is a float. If T is a float and wvl an array, the result has the length of wvl. If T is an array and wvl is a float, the result has the shape of T. Finally, if both T and wvl are arrays (T has dimension N and wvl has dimension 1), the result has the N+1 dimensions (the N dimensions of T plus the dimension of wvl).

#### Return type

float or M-D array (shape=(nWavelength), float [W/m<sup>2</sup>/sr/m]))

htrdrPy.helperFunctions.**sphere2cart**(vec)

Convert spherical to cartesian coordinates.

#### Parameters

**vec** (`numpy.ndarray`) – Spherical coordinate array (altitude [m], latitude [°], longitude [°]).

#### Returns

Cartesian coordinate array (x [m], y [m], z [m]).

#### Return type

`numpy.ndarray`



## PYTHON MODULE INDEX

### h

- `htrdrPy.data`, [5](#)
- `htrdrPy.geometry`, [20](#)
- `htrdrPy.helperFunctions`, [32](#)
- `htrdrPy.postprocess`, [28](#)
- `htrdrPy.script`, [26](#)





## B

`bandIntegratedImage()` (*htrdrPy.script.Script method*), 26

## C

`cart2sphere()` (*in module htrdrPy.helperFunctions*), 32

`cleanInputs()` (*htrdrPy.data.Data method*), 7

`combineEstimates()` (*in module htrdrPy.helperFunctions*), 32

`compositeRGB()` (*htrdrPy.script.Script method*), 26

## D

`Data` (*class in htrdrPy.data*), 5

`dplanck_dT()` (*in module htrdrPy.helperFunctions*), 32

## E

`exportGeometry()` (*htrdrPy.geometry.Geometry method*), 21

`extractMeanRadianceFromOutput()` (*htrdrPy.postprocess.Postprocess method*), 30

`extractMeanRadiances()` (*htrdrPy.postprocess.Postprocess method*), 31

## G

`Geometry` (*class in htrdrPy.geometry*), 20

`geometryFromAPIE()` (*htrdrPy.geometry.Geometry method*), 21

`getImage()` (*htrdrPy.postprocess.Postprocess method*), 31

`getImages()` (*htrdrPy.postprocess.Postprocess method*), 31

## H

`htrdrPy.data`  
module, 5

`htrdrPy.geometry`  
module, 20

`htrdrPy.helperFunctions`  
module, 32

`htrdrPy.postprocess`  
module, 28

`htrdrPy.script`

module, 26

## I

`imageRatio()` (*htrdrPy.script.Script method*), 27

## L

`loadScript()` (*in module htrdrPy.script*), 28

## M

`makeAtmosphereFrom1D()` (*htrdrPy.data.Data method*), 7

`makeAtmosphereFrom1D_PP()` (*htrdrPy.data.Data method*), 9

`makeAtmosphereFrom2D()` (*htrdrPy.data.Data method*), 10

`makeAtmosphereFrom3D()` (*htrdrPy.data.Data method*), 12

`makeFromLMDZ()` (*htrdrPy.data.Data method*), 14

`makeGroundFrom1D()` (*htrdrPy.data.Data method*), 16

`makeGroundFrom1D_PP()` (*htrdrPy.data.Data method*), 17

`makeGroundFrom2D()` (*htrdrPy.data.Data method*), 17

`makeGroundFrom3D()` (*htrdrPy.data.Data method*), 18

module

`htrdrPy.data`, 5

`htrdrPy.geometry`, 20

`htrdrPy.helperFunctions`, 32

`htrdrPy.postprocess`, 28

`htrdrPy.script`, 26

`monochromaticImage()` (*htrdrPy.script.Script method*), 27

## P

`planck()` (*in module htrdrPy.helperFunctions*), 33

`plotGeometry()` (*htrdrPy.geometry.Geometry method*), 22

`Postprocess` (*class in htrdrPy.postprocess*), 28

`processImages()` (*htrdrPy.postprocess.Postprocess method*), 31

`processSingleArrayObsSW()` (*htrdrPy.postprocess.Postprocess method*), 31

## R

reflectanceSpectrum() (*htrdrPy.script.Script method*), 27

## S

Script (*class in htrdrPy.script*), 26

setCamera() (*htrdrPy.geometry.Geometry method*), 22

setImage() (*htrdrPy.geometry.Geometry method*), 23

setSource() (*htrdrPy.geometry.Geometry method*), 23

setSpectralCumulDist() (*htrdrPy.geometry.Geometry method*), 23

setVolrad() (*htrdrPy.geometry.Geometry method*), 23

spectrum() (*htrdrPy.script.Script method*), 27

sphere2cart() (*in module htrdrPy.helperFunctions*), 33

startMultipleObsGeometry() (*htrdrPy.script.Script method*), 28

startRadBudgetGCM() (*htrdrPy.script.Script method*), 28

## V

visibleImage() (*htrdrPy.script.Script method*), 28

## W

writeInputAtmosphere() (*htrdrPy.data.Data method*), 19

writeInputGround() (*htrdrPy.data.Data method*), 20

writeInputs() (*htrdrPy.data.Data method*), 20

writeVTKfiles() (*htrdrPy.data.Data method*), 20

writeVTKfilesAtmosphere() (*htrdrPy.data.Data method*), 20

writeVTKfilesGround() (*htrdrPy.data.Data method*), 20