SMRT Documentation

Release 1.0

G. Picard, M. Sandells, H. Löwe

Sep 16, 2022

Contents

The SMRT API documentation describes the structure of the package and modules and provides detailed information on the classes and functions. It is not a practical guide for beginners to learn SMRT even though a few examples are sometimes given. We recommend to first read the tutorials <link here> and then use this API documentation as a further step to exploit SMRT in depth. SMRT extensively uses default/optional arguments in functions to provide a simple yet extendable interface. The API documentation is the only valid/up-to-date reference for these default behaviours as it is auto-generated from source. For developers who want to implement new behaviour in SMRT for their own use or for improving SMRT, we recomend to read the developer guidelines <link here> and to contact the authors of the model to discuss about the best/most generic approach to solve your problem. More documentation for improving SMRT will be prepared in the future.

The following package describes all the packages available in SMRT. The *inputs* package includes the functions to build the medium and the sensor configuration, it will include in the future any useful functions for inputs from various sources (text file, snowpack model simulations, etc). The *permittivity* package provides formulae to compute the permittivity of raw materials such as ice. The *microstructure_model* package includes all the representations of the snow micro-structure available. It provides information on the required and optional parameters of each microstrcuture_model. *interface* provides the formulation for different types of inter-layer interfaces (such as flat, rugged in the future).

The *substrate* package and *atmosphere* packages provide the lower and upper boundary conditions of the radiative transfer. Substrate can represent the soil, ice, ocean. It is worth noting that these modules describe the half-space semi-infinite media under and above the snowpack. It means they have uniform properties and especially temperature which is common practice when the focus is on the snowpack. However, for a proper fully coupled multi-layered soil-snow-atmosphere radiative transfer model, it would be necessary to describe the soil and the atmosphere as layers (exactly as the snowpack is made of snow layers) and to implement *emmodel* adequately to the soil and atmosphere.

The *emmodel* package includes all the scattering theories available in SMRT (iba, dmrt, independent spheres (Rayleigh), ...). In some case there is an inter-dependence between the choices of micro-structure and of electro-magnetic theory. For instance, `dmrt_shortrange` only works with *sticky_hard_spheres* microstructure (this is inherent to theory) and *rayleigh* would work with any microstructure model based on spheres (ie. that defines a *radius* parameter).

The *rtsolver* package includes the numerical codes that solves the radiative transfer equation.

The *core* package is where the SMRT machinery is implemented and especially the most important objects *Sensor*, *Layer*, *Snowpack*, *Model*, etc. It may be useful to understand how these objects work but it is not necessary as most of them (all) are created by helper functions which are much more convenient to use than class constructors. The only exception, which is worth exploring a bit, is *Result*. It provides useful methods to extract the result of the radiative calculation. In general, it is not recommended to modify/extend *core* for normal needs. This package does contain any science.

The *utils* package provides various useful tools to work with SMRT, but they are not strictly necessary. This package includes wrappers to some off-the-shelf models such as DMRT-QMS, HUT and MEMLS.

CHAPTER 1

smrt.inputs package

1.1 Submodules

1.2 smrt.inputs.altimeter_list module

envisat_ra2(*channel=None*)
return an Altimeter instance for the ENVISAT RA2 altimeter.

Parameters **channel** – can be 'S', 'Ku', or both. Default is both.

sentinel3_sral(*channel=None*)
return an Altimeter instance for the Sentinel 3 SRAL instrument.

Parameters **channel** – can be 'Ku' only ('C' is to be implemented)

saral_altika()
return an Altimeter instance for the Saral/AltiKa instrument.

cryosat2_lrm()
Return an altimeter instance for CryoSat-2

Parameters from https://earth.esa.int/web/eoportal/satellite-missions/c-missions/cryosat-2 Altitude from https://doi.org/10.1016/j.asr.2018.04.014 Beam width is 1.08 along track and 1.2 across track

cryosat2_sin()
Return an altimeter instance for CryoSat-2: SIN mode

Parameters from https://earth.esa.int/web/eoportal/satellite-missions/c-missions/cryosat-2 Altitude from https://doi.org/10.1016/j.asr.2018.04.014 Beam width is 1.08 along track and 1.2 across track

asiras_lam(*altitude=None*)
Return an altimeter instance for ASIRAS in Low Altitude Mode

Parameters from https://earth.esa.int/web/eoportal/airborne-sensors/asiras Beam width is 2.2 x 9.8 deg

1.3 smrt.inputs.make_medium module

The helper functions in this module are used to create snowpacks, sea-ice and other media. They are user-friendly and recommended for most usages. Extension of these functions is welcome on the condition they keep a generic structure.

The function *make_snowpack()* is the first entry point the user should consider to build a snowpack. For example:

```
from smrt import make_snowpack

sp = make_snowpack([1000], density=[300], microstructure_model='sticky_hard_spheres',␣
↪radius=[0.3e-3], stickiness=0.2)
```

creates a semi-infinite snowpack made of sticky hard spheres with radius 0.3mm and stickiness 0.2. The `Snowpack` object is in the *sp* variable.

Note that *make_snowpack* is directly imported from *smrt* instead of *smrt.inputs.make_medium*. This feature is for convenience.

make_medium(*data*, *surface=None*, *interface=None*, *substrate=None*, ***kwargs*)

build a multi-layered medium using a pandas DataFrame (or a dict that can be transformed into a DataFrame) and optinal arguments. The 'medium' column (or key) in data indicates the medium type: 'snow' or 'ice'. If not given, it defaults to 'snow'. 'data' must contain enough information to build either a snowpack or an ice_column. The minimum requirements are:

- for a snowpack: ('z' or 'thickness'), 'density', 'microstructure_model' and the arguments required by the microstructural_model.
- for a ice column: ice_type, ('z' or 'thickness'), 'temperature', 'salinity', 'microstructure_model' and the arguments required by

the microstructural_model.

When reading a dataframe from disk for instance, it is convenient to use df.rename(columns={...}) to map the column names of the file to the column names required by SMRT.

if 'z' is given, the thickness is deduced using `compute_thickness_from_z()`.

** Warning **: Using this function is a bit dangerous as any unrecognized column names are silently ignored. For instance, a column named 'Temperature' is ignore (due to the uppercase), and the temperature in the snowpack will be set to its default value (273.15 K). This issue applies to any optional argument. Double ckeck the spelling of the columns.

**** Note **: *make_medium* create laters using all the columns in the dataframe. It means that any column name becomes a**

the layer objects, even if not recognized/used by SMRT. This can be seen as an interesting feature to store information in layers, but this is also dangerous if column names collide with internal layer attributes or method names. It is recommended to clean the dataframe (with df.drop(columns=[...])) before calling make_medium.

make_snowpack(*thickness*, *microstructure_model*, *density*, *interface=None*, *surface=None*, *substrate=None*, *atmosphere=None*, ***kwargs*)

build a multi-layered snowpack. Each parameter can be an array, list or a constant value.

Parameters **thickness** – thicknesses of the layers in meter (from top to bottom). The last layer thickness can be "numpy.inf"

for a semi-infinite layer. :param microstructure_model: microstructure_model to use (e.g. sticky_hard_spheres or independent_sphere or exponential). :param surface: type of surface interface, flat/fresnel is the default. If surface and interface are both set, the interface must be a constant refering to all the "internal" interfaces. :param interface: type of interface, flat/fresnel is the default. It is usually a string for the interfaces without parameters (e.g. Flat or Transparent) or is created with *make_interface()* in more complex cases. Interface can be a

constant or a list. In the latter case, its length must be the same as the number of layers, and interface[0] refers to the surface interface. :param density: densities of the layers. :param substrate: set the substrate of the snowpack. Another way to add a substrate is to use the + operator (e.g. snowpack + substrate). :param **kwargs: All the other parameters (temperature, microstructure parameters, emmodel, etc.) are given as optional arguments (e.g. temperature=[270, 250]). They are passed for each layer to the function *make_snow_layer()*. Thus, the documentation of this function is the reference. It describes precisely the available parameters. The microstructure parameter(s) depend on the microstructure_model used and is documented in each microstructure_model module.

e.g.:

```
sp = make_snowpack([1, 10], "exponential", density=[200,300], temperature=[240,␣
↪250], corr_length=[0.2e-3, 0.3e-3])
```

make_snow_layer(*layer_thickness*, *microstructure_model*, *density*, *temperature=273.15*, *ice_permittivity_model=None*, *background_permittivity_model=1.0*, *volumetric_liquid_water=None*, *liquid_water=None*, *salinity=0*, *medium='snow'*, ***kwargs*)

Make a snow layer for a given microstructure_model (see also *make_snowpack()* to create many layers). The microstructural parameters depend on the microstructural model and should be given as additional arguments to this function. To know which parameters are required or optional, refer to the documentation of the specific microstructure model used.

Parameters

- **layer_thickness** – thickness of snow layer in m.
- **microstructure_model** – module name of microstructure model to be used.
- **density** – density of snow layer in kg m $^{-3}$. Includes the ice and water phases.
- **temperature** – temperature of layer in K.
- **ice_permittivity_model** – permittivity formulation of the scatterers (default is ice_permittivity_matzler87).
- **background_permittivity_model** – permittivity formulation for the background (default is air).
- **volumetric_liquid_water** – volume of liquid water with respect to the volume of snow (default=0).
- **liquid_water** – May be depreciated in the future (use instead volumetric_liquid_water): volume of liquid water

with respect to ice+water volume (default=0). liquid_water = water_volume / (ice_volume + water_volume). :param salinity: salinity in kg/kg, for using PSU as unit see PSU constant in smrt module (default = 0). :param medium: indicate which medium the layer is made of ("snow" is a default). It is used when emmodel is a dictionary mapping from medium to emmodels in *make_model()* :param kwargs: other microstructure parameters are given as optional arguments (in Python words) but may be required (in SMRT words). See the documentation of the microstructure model.

Returns *SnowLayer* instance

class SnowLayer(**args*, *density=None*, *volumetric_liquid_water=None*, *liquid_water=None*, ***kwargs*)

Bases: *smrt.core.layer.Layer*

Specialized Layer class for snow. It deals with the calculation of the frac_volume and the liquid_water from density and volumetric_liquid_water. Alternatively it is possible to set liquid_water directly but this is not recommended anymore.

update(*density=None*, *volumetric_liquid_water=None*, *liquid_water=None*, ***kwargs*)

update the density and/or volumetric_liquid_water. This method must be used every time density and/or

volumetric_liquid_water are changed. Setting directly the corresponding attributes of the Layer object raises an error because a recalculation of the frac_volume and liquid_volume is necessary every time one of these variables is changed.

static compute_frac_volumes(*density*, *volumetric_liquid_water=None*, *liquid_water=None*)
compute and return the fractional volumes: - frac_volume =(ice+water) / (ice+water+air) - liquid_water =(water) / (ice+water)

make_ice_column(*ice_type*, *thickness*, *temperature*, *microstructure_model*, *brine_inclusion_shape='spheres'*, *salinity=0.0*, *brine_volume_fraction=None*, *brine_permittivity_model=None*, *ice_permittivity_model=None*, *saline_ice_permittivity_model=None*, *porosity=0*, *density=None*, *add_water_substrate=True*, *surface=None*, *interface=None*, *substrate=None*, *atmosphere=None*, ***kwargs*)
Build a multi-layered ice column. Each parameter can be an array, list or a constant value.

ice_type variable determines the type of ice, which has a big impact on how the medium is modelled and the parameters: - First year ice is modelled as scattering brines embedded in a pure ice background - Multi year ice is modelled as scattering air bubbles in a saline ice background (but brines are non-scattering in this case). - Fresh ice is modelled as scattering air bubbles in a pure ice background (but brines are non-scattering in this case).

First-year and multi-year ice is equivalent only if scattering and porosity are nulls. It is important to understand that in multi-year ice scattering by brine pockets is neglected because scattering is due to air bubbles and the emmodel implemented up to now are not able to deal with three-phase media.

Parameters

- **ice_type** – Ice type. Options are "firstyear", "multiyear", "fresh"
- **thickness** – thicknesses of the layers in meter (from top to bottom). The last layer thickness can be "numpy.inf"

for a semi-infinite layer. :param temperature: temperature of ice/water in K :param brine_inclusion_shape: assumption for shape of brine inclusions. So far, "spheres" or "random_needles" (i.e. elongated ellipsoidal inclusions), and "mix" (a mix of the two) are implemented. :param salinity: salinity of ice/water in kg/kg (see PSU constant in smrt module). Default is 0. If neither salinity nor brine_volume_fraction are given, the ice column is considered to consist of fresh water ice. :param brine_volume_fraction: brine / liquid water fraction in sea ice, optional parameter, if not given brine volume fraction is calculated from temperature and salinity in ~.smrt.permittivity.brine_volume_fraction :param density: density of ice layer in kg m $^{-3}$:param porosity: porosity of ice layer (0 - 1). Default is 0. :param add_water_substrate: Adds a substrate made of water below the ice column. Possible arguments are True (default) or False. If True looks for ice_type to determine if a saline or fresh water layer is added and/or uses the optional arguments 'water_temperature', 'water_salinity' of the water substrate. :param surface: type of surface interface, flat/fresnel is the default. If surface and interface are both set, the interface must be a constant refering to all the "internal" interfaces. :param interface: type of interface, flat/fresnel is the default. It is usually a string for the interfaces without parameters (e.g. Flat or Transparent) or is created with *make_interface()* in more complex cases. Interface can be a constant or a list. In the latter case, its length must be the same as the number of layers, and interface[0] refers to the surface interface. :param substrate: if add_water_substrate is False, the substrate can be prescribed with this argument.

All the other optional arguments are passed for each layer to the function *make_ice_layer()*. The documentation of this function describes in detail the parameters used/required depending on ice_type.

make_ice_layer(*ice_type*, *layer_thickness*, *temperature*, *salinity*, *microstructure_model*, *brine_inclusion_shape='spheres'*, *brine_volume_fraction=None*, *porosity=0*, *density=None*, *brine_permittivity_model=None*, *ice_permittivity_model=None*, *saline_ice_permittivity_model=None*, *medium='ice'*, ***kwargs*)
Make an ice layer for a given microstructure_model (see also *make_ice_column()* to create many layers). The microstructural parameters depend on the microstructural model and should be given as additional argu-

ments to this function. To know which parameters are required or optional, refer to the documentation of the specific microstructure model used.

Parameters

- **ice_type** – Assumed ice type
- **layer_thickness** – thickness of ice layer in m
- **temperature** – temperature of layer in K
- **salinity** – (firstyear and multiyear) salinity in kg/kg (see PSU constant in smrt module)
- **brine_inclusion_shape** – (firstyear and multiyear) assumption for shape of brine inclusions (so far,

"spheres" and "random_needles" (i.e. elongated ellipsoidal inclusions), and "mix_spheres_needles" are implemented) :param brine_volume_fraction: (firstyear and multiyear) brine / liquid water fraction in sea ice, optional parameter, if not given brine volume fraction is calculated from temperature and salinity in ~.smrt.permittivity.brine_volume_fraction :param density: (multiyear) density of ice layer in kg m $^{-3}$. If not given, density is calculated from temperature, salinity and ice porosity. :param porosity: (mutliyear and fresh) air porosity of ice layer (0..1). Default is 0. :param ice_permittivity_model: (all) pure ice permittivity formulation (default is ice_permittivity_matzler06) :param brine_permittivity_model: (firstyear and multiyear) brine permittivity formulation (default is brine_permittivity_stogryn85) :param saline_ice_permittivity_model: (multiyear) model to mix ice and brine. The default uses polder van staten and ice_permittivity_model and brine_permittivity_model. It is highly recommanded to use the default. :param kwargs: other microstructure parameters are given as optional arguments (in Python words) but may be required (in SMRT words). :param medium: indicate which medium the layer is made of ("ice" is a default). It is used when emmodel is a dictionary mapping from medium to emmodels in *make_model()*

See the documentation of the microstructure model.

Returns `Layer` instance

make_water_body(*layer_thickness=1000*, *temperature=273*, *salinity=0*, *water_permittivity_model=None*, *surface=None*, *atmosphere=None*, *substrate=None*)

Make a water body with a single layer of water at given temperature and salinity.

Note that water is a very strong absorber even fresh water, it is unlikely that the layers under a water body could be seen by microwaves. If really needed anyway, a multi-layer water body or

a water layer on another medium (e.g. ice) can be build using the addition operator.

Parameters

- **layer_thickness** – thickness of ice layer in m
- **temperature** – temperature of layer in K
- **salinity** – salinity in kg/kg (see PSU constant in smrt module)
- **water_permittivity_model** – water permittivity formulation (default is seawater_permittivity_klein76)
- **surface** – type of surface interface. Flat surface (Fresnel coefficient) is the default.
- **substrate** – the substrate under the water layer.

make_water_layer(*layer_thickness*, *temperature=273*, *salinity=0*, *water_permittivity_model=None*, ***kwargs*)

Make a water layer at given temperature and salinity.

Parameters

- **layer_thickness** – thickness of ice layer in m
- **temperature** – temperature of layer in K
- **salinity** – salinity in kg/kg (see PSU constant in smrt module)
- **water_permittivity_model** – water permittivity formulation (default is seawater_permittivity_klein76)

water_parameters(*ice_type*, ***kwargs*)
Make a semi-infinite water layer.

Parameters **ice_type** – ice_type is used to determine if a saline or fresh water layer is added

Optional arguments are 'water_temperature', 'water_salinity' and 'water_depth' of the water layer.

bulk_ice_density(*temperature*, *salinity*, *porosity*)
Computes bulk density of sea ice (in kg m $^{-3}$), when considering the influence from brine, solid salts, and air bubbles in the ice. Formulation from Cox & Weeks (1983): Equations for determining the gas and brine volumes in sea ice samples, J Glac. Developed for temperatures between -2–30°C. For higher temperatures (>2°C) is used the formulation from Lepparanta & Manninen (1988): The brine and gas content of sea ice with attention to low salinities and high temperatures.

Parameters

- **temperature** – Temperature in K
- **salinity** – salinity in kg/kg (see PSU constant in smrt module)
- **porosity** – Fractional volume of air inclusions (0..1)

Returns Density of ice mixture in kg m $^{-3}$

make_generic_stack(*thickness*, *temperature=273*, *ks=0*, *ka=0*, *effective_permittivity=1*, *interface=None*, *substrate=None*, *atmosphere=None*)
build a multi-layered medium with prescribed scattering and absorption coefficients and effective permittivity. Must be used with presribed_kskaeps emmodel.

Parameters **thickness** – thicknesses of the layers in meter (from top to bottom). The last layer thickness can be "numpy.inf" for

a semi-infinite layer. :param temperature: temperature of layers in K :param ks: scattering coefficient of layers in m^-1 :param ka: absorption coefficient of layers in m^-1 :param interface: type of interface, flat/fresnel is the default

make_generic_layer(*layer_thickness*, *ks=0*, *ka=0*, *effective_permittivity=1*, *temperature=273*)
Make a generic layer with prescribed scattering and absorption coefficients and effective permittivity. Must be used with presribed_kskaeps emmodel.

Parameters

- **layer_thickness** – thickness of ice layer in m
- **temperature** – temperature of layer in K
- **ks** – scattering coefficient of layers in m^-1
- **ka** – absorption coefficient of layers in m^-1

Returns `Layer` instance

make_atmosphere(*atmosphere_model*, ***kwargs*)
make a atmospheric single-layer using the prescribed atmosphere model. Warning: this function is subject to change in the future when refactoring how SMRT deals with atmosphere.

Parameters

- **atmosphere_model** – the name of the model to use. The available models are in smrt.atmosphere.
- ****kwargs** – all the parameters used by the atmosphere_model.

compute_thickness_from_z(*z*)

Compute the thickness of layers given the elevation z. Whatever the sign of z, the order *MUST* be from the topmost layer t
lowermost.

Several situation are accepted and interpretated as follows: - z is positive and decreasing. The first value is the height of the surface about the ground (z=0) and z represents the top elevation of each layer. This is typical of the seasonal snowpack. - z is negative and decreasing. The first value is the elevation of the bottom of the first layer with respect to the surface (z=0). This is typical of a snowpack on ice-sheet. - z is positive and increasing. The first value is the depth of the bottom of the first layer with respect to the surface. This is typical of a snowpack on ice-sheet. - other case, when z is not monoton or is increasing with negative value raises an error.

Because z indicate the top or the bottom of a layer depending whether z=0 is the ground or the surface, the value 0 can never be in z. This raises an error.

1.4 smrt.inputs.make_soil module

This module provides a function to build soil model and provides some soil permittivity formulae.

To create a substrate, use/implement an helper function such as `make_soil()`. This function is able to automatically load a specific soil model and provides some soil permittivity formulae as well.

Examples:

```
from smrt import make_soil
soil = make_soil("soil_wegmuller", "dobson85", moisture=0.2, sand=0.4, clay=0.3,␣
↪drymatter=1100, roughness_rms=1e-2)
```

It is recommand to first read the documentation of `make_soil()` and then explore the different types of soil models.

make_soil(*substrate_model*, *permittivity_model*, *temperature*, *moisture=None*, *sand=None*, *clay=None*, *drymatter=None*, ***kwargs*)

Construct a soil instance based on a given surface electromagnetic model, a permittivity model and parameters

Parameters

- **substrate_model** – name of substrate model, can be a class or a string. e.g. fresnel, wegmuller...
- **permittivity_model** – permittivity_model to use. Can be a name ("hut_epss", "dobson85", "montpetit2008"), a function of

frequency and temperature or a complex value. :param moisture: soil moisture in m:sup:*3* m:sup:*-3* to compute the permittivity. This parameter is used depending on the permittivity_model. :param sand: soil relative sand content. This parameter is used or not depending on the permittivity_model. :param clay: soil relative clay content. This parameter is used or not depending on the permittivity_model. :param drymatter: soil content in dry matter in kg m:sup:*-3*. This parameter is used or not depending on the permittivity_model.

Parameters ****kwargs** – geometrical parameters depending on the substrate_model. Refer to the document of each model to see the

list of required and optional parameters. Usually, it is roughness_rms, corr_length, ...

Usage example:

:: TOTEST: bottom = substrate.make('Flat', permittivity_model=complex('6-0.5j')) TOTEST: bottom = substrate.make('Wegmuller', permittivity_model='soil', roughness_rms=0.25, moisture=0.25)

soil_dielectric_constant_dobson (*frequency*, *tempK*, *SM*, *S*, *C*)

soil_dielectric_constant_hut (*frequency*, *tempK*, *SM*, *sand*, *clay*, *dm_rho*)

soil_dielectric_constant_monpetit2008 (*frequency*, *temperature*)

Soil dielectric constant formulation based on the formulation Montpetit et al. 2018. The formulation is only valid for below-frrezing point temperature.

Reference: Montpetit, B., Royer, A., Roy, A., & Langlois, A. (2018). In-situ passive microwave emission model parameterization of sub-arctic frozen organic soils. Remote Sensing of Environment, 205, 112–118. https://doi.org/10.1016/j.rse.2017.10.033

1.5 smrt.inputs.make_substrate module

1.6 smrt.inputs.sensor_list module

The sensor configuration includes all the information describing the sensor viewing geometry (incidence, ...) and operating parameters (frequency, polarization, ...). The easiest and recommended way to create a `Sensor` instance is to use one of the convenience functions listed below. The generic functions *`passive()`* and *`active()`* should cover all the usages, but functions for specific sensors are more convenient. See examples in the functions documentation below. We recommend to add new sensors/functions here and share your file to be included in SMRT.

passive (*frequency*, *theta*, *polarization=None*, *channel_map=None*, *name=None*)

Generic configuration for passive microwave sensor.

Return a `Sensor` for a microwave radiometer with given frequency, incidence angle and polarization

Parameters

- **frequency** – frequency in Hz
- **theta** – viewing angle or list of viewing angles in degrees from vertical. Note that some RT solvers compute all viewing angles whatever this configuration because it is internally needed part of the multiple scattering calculation. It it therefore often more efficient to call the model once with many viewing angles instead of calling it many times with a single angle.
- **polarization** (*list of characters*) – H and/or V polarizations. Both polarizations is the default. Note that most RT solvers compute all the polarizations whatever this configuration because the polarizations are coupled in the RT equation.
- **channel_map** (*dict*) – map channel names (keys) to configuration (values). A configuration is a dict with frequency, polarization and other such parameters to be used by Result to select the results.
- **name** (*string*) – name of the sensor

Returns `Sensor` instance

Usage example:

```
from smrt import sensor
radiometer = sensor.passive(18e9, 50)
radiometer = sensor.passive(18e9, 50, "V")
radiometer = sensor.passive([18e9,36.5e9], [50,55], ["V","H"])
```

active(*frequency*, *theta_inc*, *theta=None*, *phi=None*, *polarization_inc=None*, *polarization=None*, *channel_map=None*, *name=None*)

Configuration for active microwave sensor.

Return a `Sensor` for a radar with given frequency, incidence and viewing angles and polarization

If polarizations are not specified, quad-pol is the default (VV, VH, HV and HH). If the angle of incident radiation is not specified, *backscatter* will be simulated

Parameters

- **frequency** – frequency in Hz
- **theta_inc** – incident angle in degrees from the vertical
- **theta** – viewing zenith angle in degrees from the vertical. By default, it is equal to theta_inc which corresponds to the backscatter direction
- **phi** – viewing azimuth angle in degrees from the incident direction. By default, it is pi which corresponds to the backscatter direction
- **polarization_inc** (*list of 1-character strings*) – list of polarizations of the incidence wave ('H' or 'V' or both.)
- **polarization** (*list of 1-character strings*) – list of viewing polarizations ('H' or 'V' or both)
- **channel_map** (*dict*) – map channel names (keys) to configuration (values). A configuration is a dict with frequency, polarization and other such parameters to be used by Result to select the results.
- **name** (*string*) – name of the sensor

Returns `Sensor` instance

Usage example:

```
from smrt import sensor
scatterometer = sensor.active(frequency=18e9, theta_inc=50)
scatterometer = sensor.active(18e9, 50, 50, 0, "V", "V")
scatterometer = sensor.active([18e9,36.5e9], theta=50, theta_inc=50, polarization_
↪inc=["V", "H"], polarization=["V", "H"])
```

amsre(*channel=None*, *frequency=None*, *polarization=None*, *theta=55*)

Configuration for AMSR-E sensor.

This function can be used to simulate all 12 AMSR-E channels i.e. frequencies of 6.925, 10.65, 18.7, 23.8, 36.5 and 89 GHz at both polarizations H and V. Alternatively single channels can be specified with 3-character identifiers. 18 and 19 GHz can be used interchangably to represent 18.7 GHz, similarly either 36 and 37 can be used to represent the 36.5 GHz channel. Note that if you need both H and V polarization (at 37 GHz for instance), use channel="37" instead of channel=["37V", "37H"] as this will result in a more efficient simulation, because most rtsolvers anyway compute both polarizations in one shot.

Parameters **channel** (*3-character string*) – single channel identifier

Returns `Sensor` instance

Usage example:

```
from smrt import sensor
radiometer = sensor.amsre()  # Simulates all channels
radiometer = sensor.amsre('36V')  # Simulates 36.5 GHz channel only
radiometer = sensor.amsre('06H')  # 6.925 GHz channel
```

amsr2 (*channel=None*, *frequency=None*, *polarization=None*, *theta=55*)
Configuration for AMSR-2 sensor.

This function can be used to simulate all 14 AMSR2 channels i.e. frequencies of 6.925, 10.65, 18.7, 23.8, 36.5 and 89 GHz at both polarizations H and V. Alternatively single channels can be specified with 3-character identifiers. 18 and 19 GHz can be used interchangably to represent 18.7 GHz, similarly either 36 and 37 can be used to represent the 36.5 GHz channel. Note that if you need both H and V polarization (at 37 GHz for instance), use channel="37" instead of channel=["37V", "37H"] as this will result in a more efficient simulation, because most rtsolvers anyway compute both polarizations in one shot.

Parameters **channel** (*3-character string*) – single channel identifier

Returns `Sensor` instance

Usage example:

```
from smrt import sensor
radiometer = sensor.amsre()  # Simulates all channels
radiometer = sensor.amsre('36V')  # Simulates 36.5 GHz channel only
radiometer = sensor.amsre('06H')  # 6.925 GHz channel
```

cimr (*channel=None*, *frequency=None*, *polarization=None*, *theta=55*)
Configuration for AMSR-2 sensor.

This function can be used to simulate all 10 CIMR channels i.e. frequencies of 1.4, 6.9, 10.6, 18.7, 36.5 GHz at both polarizations H and V. Alternatively single channels can be specified with 3-character identifiers. 18 and 19 GHz can be used interchangably to represent 18.7 GHz, similarly either 36 and 37 can be used to represent the 36.5 GHz channel. Note that if you need both H and V polarization (at 37 GHz for instance), use channel="37" instead of channel=["37V", "37H"] as this will result in a more efficient simulation, because most rtsolvers anyway compute both polarizations in one shot.

Parameters **channel** (*3-character string*) – single channel identifier

Returns `Sensor` instance

common_conical_pmw (*sensor_name*, *frequency_dict*, *channel=None*, *frequency=None*, *polarization=None*, *theta=55*, *name=None*)

quikscat (*channel=None*, *theta=None*)
Configuration for quikscat sensor.

This function can be used to simulate the 4 QUIKSCAT channels i.e. incidence angles 46° and 54° and HH and VV polarizations. Alternatively a subset of these channels can be specified with 4-character identifiers with polarization first .e.g. HH46, VV54

Parameters **channel** (*4-character string*) – single channel identifier

Returns `Sensor` instance

ascat (*theta=None*)
Configuration for ASCAT on ENVISAT sensor.

This function returns a sensor at 5.255 GHz (C-band) and VV polarization. The incidence angle can be chosen or is by defaut from 25° to 65° every 5°

Parameters **theta** (*float or sequence*) – incidence angle (between 25 and 65° in principle)

Returns `Sensor` instance

sentinel1 (*theta=None*)
Configuration for C-SAR on Sentinel 1.

This function return a sensor at 5.405 GHz (C-band). The incidence angle can be chosen or is by defaut from 20 to 45° by step of 5°

Parameters **theta** (*float or sequence*) – incidence angle

Returns Sensor instance

smos (*theta=None*)
Configuration for MIRAS on SMOS.

This function returns a passive sensor at 1.41 GHz (L-band). The incidence angle can be chosen or is by defaut from 0 to 60° by step of 5°

Parameters **theta** (*float or sequence*) – incidence angle

Returns Sensor instance

smap (*mode*, *theta=40*)
Configuration for the passive (mode=P) and active (mode=A) sensor on smap

This function returns either a passive sensor at 1.4 GHz (L-band) sensor or an active sensor at 1.26 GHz. The incidence angle is 40°.

filter_channel_map (*channel_map*, *channel*)

extract_configuration (*channel_map*)

1.7 smrt.inputs.test_make_medium module

test_make_snowpack()

test_make_snowpack_surface_interface()

test_make_snowpack_interface()

test_make_snowpack_surface_and_list_interface()

test_make_snowpack_with_scalar_thickness()

test_make_snowpack_array_size()

test_make_lake_ice()

test_make_medium()

test_make_snowpack_volumetric_liquid_water()

test_update_volumetric_liquid_water()

test_snow_set_readonly()

1.8 smrt.inputs.test_make_substrate module

1.9 smrt.inputs.test_sensor_list module

test_map_channel19_to_dictionary()

test_map_channel37_to_dictionary()

test_amsre_theta_is_55()

test_amsre_channel_recognized()

test_map_channel06_to_dictionary()

test_map_channel07_to_dictionary()

test_amsr2_theta_is_55()

test_cimr_channel01_to_dictionary()

test_cimr_is_55()

1.10 Module contents

This package includes modules to create the medium and sensor configuration required for the simulations. The recommended way to build these objects:

```
from smrt import make_snowpack, sensor_list

sp = make_snowpack([1000], density=[300], microstructure_model='sticky_hard_spheres',␣
↪radius=[0.3e-3], stickiness=0.2)

radiometer = sensor_list.amsre()
```

Note that the function *make_snowpack()* and the module *sensor_list* is directly imported from smrt, which is convenient but they effectively lie in the package *smrt.inputs*. They could be imported using the full path as follows:

```
from smrt.inputs.make_medium import make_snowpack
from smrt.inputs import sensor_list

sp = make_snowpack([1000], density=[300], microstructure_model='sticky_hard_spheres',␣
↪radius=[0.3e-3], stickiness=0.2)

radiometer = sensor_list.amsre()
```

Extension of the modules in the *inputs* package is welcome. This is as simple as adding new functions in the modules (e.g. in *sensor_list*) or adding a new modules (e.g. *my_make_medium.py*) in this package and use the full path import.

CHAPTER 2

smrt.permittivity package

2.1 Submodules

2.2 smrt.permittivity.brine module

brine_conductivity(*temperature*)
computes ionic conductivity of dissolved salts, Stogryn and Desargant, 1985

Parameters **temperature** – thermometric temperature [K]

brine_relaxation_time(*temperature*)
computes relaxation time of brine, Stogryn and Desargant, 1985

Parameters **temperature** – thermometric temperature [K]

static_brine_permittivity(*temperature*)
computes static dielectric constant of brine, Stogryn and Desargant, 1985

Parameters **temperature** – thermometric temperature [K]

calculate_brine_salinity(*temperature*)
Computes the salinity of brine (in ppt) for a given temperature (Cox and Weeks, 1975)

Parameters **temperature** – snow temperature in K

:return salinity_brine in ppt

permittivity_high_frequency_limit(*temperature*)
computes permittivity.

Parameters **temperature** – ice or snow temperature in K

brine_volume(*temperature*, *salinity*, *porosity=0*, *bulk_density=None*)
computes brine volume fraction using coefficients from Cox and Weeks (1983): 'Equations for determining the gas and brine volumes in sea-ice samples', J. of Glac. if ice temperature is below -2 deg C or coefficients determined by Lepparanta and Manninen (1988): 'The brine and gas content of sea ice with attention to low salinities and high temperatures' for warmer temperatures.

Parameters

- **temperature** – ice temperature in K
- **salinity** – salinity of ice in kg/kg (see PSU constant in smrt module)
- **porosity** – fractional air volume in ice (0..1). Default is 0.
- **bulk_density** – density of bulk ice in kg m $^{-3}$

calculate_freezing_temperature (*salinity*)

calculates temperature at which saline water freezes using polynomial fits of the Gibbs function given in TEOS-10: The international thermodynamic equation of seawater - 2010 ('http://www.teos-10.org/pubs/TEOS-10_Manual.pdf). The error of this fit ranges between -5e-4 K and 6e-4 K when compared with the temperature calculated from the exact in-situ freezing temperature, which is found by a Newton-Raphson iteration of the equality of the chemical potentials of water in seawater and in ice.

Parameters **salinity** – salinity of ice in kg/kg (see PSU constant in smrt module)

2.3 smrt.permittivity.generic_mixing_formula module

This module contains functions that are not tied to a particular electromagnetic model and are available to be imported by any electromagnetic model. It is the responsibility of the developer to ensure these functions, if used, are appropriate and consistent with the physics of the electromagnetic model.

depolarization_factors (*length_ratio=None*)

Calculates depolarization factors for use in effective permittivity models. These are a measure of the anisotropy of the snow. Default is spherical isotropy.

Parameters **length_ratio** – [Optional] ratio of microstructure length measurement in x/y direction to z-direction [unitless].

Returns [x, y, z] depolarization factor array

Usage example:

```
from smrt.permittivity.generic_mixing_formula import depolarization_factors
depol_xyz = depolarization_factors(length_ratio=1.2)
depol_xyz = depolarization_factors()
```

polder_van_santen (*frac_volume*, *e0=1*, *eps=3.185*, *depol_xyz=None*, *length_ratio=None*, *inclusion_shape=None*, *mixing_ratio=1*)

Calculates effective permittivity of snow by solution of quadratic Polder Van Santen equation for spherical inclusion.

Parameters

- **frac_volume** – Fractional volume of inclusions
- **e0** – Permittivity of background (default is 1)
- **eps** – Permittivity of scattering material (default is 3.185 to compare with MEMLS)
- **depol_xyz** – [Optional] Depolarization factors, spherical isotropy is default. It is not taken into account here.
- **length_ratio** – Length_ratio. Used to estimate depolarization factors when they are not given.

- **inclusion_shape** – Assumption for shape(s) of brine inclusions. Can be a string for single shape, or a list/tuple/dict of strings for mixture of shapes. So far, we have the following shapes: "spheres" and "random_needles" (i.e. randomly-oriented elongated ellipsoidal inclusions). If the argument is a dict, the keys are the shapes and the values are the mixing ratio. If it is a list, the mixing_ratio argument is required.
- **mixing_ratio** – The mixing ratio of the shapes. This is only relevant when inclusion_shape is a list/tuple. Mixing ratio must be a sequence with length len(inclusion_shape)-1. The mixing ratio of the last shapes is deduced as the sum of the ratios must equal to 1.

Returns Effective permittivity

Usage example:

```
from smrt.permittivity.generic_mixing_formula import polder_van_santen
effective_permittivity = polder_van_santen(frac_volume, e0, eps)

# for a mixture of 30% spheres and 70% needles
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_shape={
↪"spheres": 0.3, "random_needles": 0.7})
# or
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_shape=(
↪"spheres", "random_needles"), mixing_ratio=0.3)
```

Todo: Extend Polder Van Santen model to account for ellipsoidal inclusions

---

bruggeman(*frac_volume*, *e0=1*, *eps=3.185*, *depol_xyz=None*, *length_ratio=None*, *inclusion_shape=None*, *mixing_ratio=1*)

Calculates effective permittivity of snow by solution of quadratic Polder Van Santen equation for spherical inclusion.

Parameters

- **frac_volume** – Fractional volume of inclusions
- **e0** – Permittivity of background (default is 1)
- **eps** – Permittivity of scattering material (default is 3.185 to compare with MEMLS)
- **depol_xyz** – [Optional] Depolarization factors, spherical isotropy is default. It is not taken into account here.
- **length_ratio** – Length_ratio. Used to estimate depolarization factors when they are not given.
- **inclusion_shape** – Assumption for shape(s) of brine inclusions. Can be a string for single shape, or a list/tuple/dict of strings for mixture of shapes. So far, we have the following shapes: "spheres" and "random_needles" (i.e. randomly-oriented elongated ellipsoidal inclusions). If the argument is a dict, the keys are the shapes and the values are the mixing ratio. If it is a list, the mixing_ratio argument is required.
- **mixing_ratio** – The mixing ratio of the shapes. This is only relevant when inclusion_shape is a list/tuple. Mixing ratio must be a sequence with length len(inclusion_shape)-1. The mixing ratio of the last shapes is deduced as the sum of the ratios must equal to 1.

Returns Effective permittivity

Usage example:

```
from smrt.permittivity.generic_mixing_formula import polder_van_santen
effective_permittivity = polder_van_santen(frac_volume, e0, eps)

# for a mixture of 30% spheres and 70% needles
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_shape={
↪"spheres": 0.3, "random_needles": 0.7})
# or
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_shape=(
↪"spheres", "random_needles"), mixing_ratio=0.3)
```

Todo: Extend Polder Van Santen model to account for ellipsoidal inclusions

polder_van_santen_three_spherical_components (*f1*, *f2*, *eps0*, *eps1*, *eps2*)
Calculates effective permittivity using Polder and van Santen with three components assuming spherical inclusions

Parameters

- **f1** – fractional volume of component 1
- **f2** – fractional volume of component 2
- **eps0** – permittivity of material 0
- **eps1** – permittivity of material 1
- **eps2** – permittivity of material 2

polder_van_santen_three_components (*f1*, *f2*, *eps0*, *eps1*, *eps2*, *A1*, *A2*)
Calculates effective permittivity using Polder and van Santen with three components

Parameters

- **f1** – fractional volume of component 1
- **f2** – fractional volume of component 2
- **eps0** – permittivity of material 0
- **eps1** – permittivity of material 1
- **eps2** – permittivity of material 2
- **A1** – depolarization factor for material 1
- **A2** – depolarization factor for material 2

maxwell_garnett (*frac_volume*, *e0*, *eps*, *depol_xyz=None*, *inclusion_shape=None*, *length_ratio=None*)
Calculates effective permittivity using Maxwell-Garnett equation.

Parameters

- **frac_volume** – Fractional volume of snow
- **e0** – Permittivity of background (no default, must be provided)
- **eps** – Permittivity of scattering material (no default, must be provided)
- **depol_xyz** – [Optional] Depolarization factors, spherical isotropy is default. It is not taken into account here.
- **length_ratio** – Length_ratio. Used to estimate depolarization factors when they are not given.

- **inclusion_shape** – Assumption for shape(s) of brine inclusions. Can be a string for single shape, or a list/tuple/dict of strings for mixture of shapes. So far, we have the following shapes: "spheres" and "random_needles" (i.e. randomly-oriented elongated ellipsoidal inclusions). If the argument is a dict, the keys are the shapes and the values are the mixing ratio. If it is a list, the mixing_ratio argument is required.

Returns random orientation effective permittivity

Usage example:

```
# If used by electromagnetic model module:
from .commonfunc import maxwell_garnett
effective_permittivity = maxwell_garnett(frac_volume=0.2,
                                         e0=1,
                                         eps=3.185,
                                         depol_xyz=[0.3, 0.3, 0.4])

# If accessed from elsewhere, use absolute import
from smrt.emmodel.commonfunc import maxwell_garnett
```

maxwell_garnett_for_spheres (*frac_volume*, *e0*, *eps*)
Calculates effective permittivity using Maxwell-Garnett equation assuming spherical inclusion. This function is essentially an optimized version of py:func:*maxwell_garnett*.

2.4 smrt.permittivity.ice module

ice_permittivity_maetzler06 (*frequency*, *temperature*)
Calculates the complex ice dielectric constant depending on the frequency and temperature

Based on Mätzler, C. (2006). Thermal Microwave Radiation: Applications for Remote Sensing p456-461 This is the default model used in smrt.inputs.make_medium.make_snow_layer().

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

Returns Complex permittivity of pure ice

Usage example:

```
from smrt.permittivity.ice import ice_permittivity_maetzler06
eps_ice = ice_permittivity_maetzler06(frequency=18e9, temperature=270)
```

Note: Ice permittivity is automatically calculated in smrt.inputs.make_medium.make_snow_layer() and is not set by the electromagnetic model module. An alternative to ice_permittivity_maetzler06 may be specified as an argument to the make_snow_layer function. The usage example is provided for external reference or testing purposes.

---

ice_permittivity_maetzler98 (*frequency*, *temperature*)
computes permittivity of ice (accounting for ionic impurities in ice?), equations from Hufford (1991) as given in Maetzler (1998): 'Microwave properties of ice and snow', in B. Schmitt et al. (eds.): 'Solar system ices', p. 241-257, Kluwer.

Parameters

- **temperature** – ice temperature in K
- **frequency** – Frequency in Hz

ice_permittivity_maetzler87 (*frequency*, *temperature*)

Calculates the complex ice dielectric constant depending on the frequency and temperature

Based on Mätzler, C. and Wegmüller (1987). Dielectric properties of fresh-water ice at microwave frequencies. J. Phys. D: Appl. Phys. 20 (1987) 1623-1630.

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

Returns Complex permittivity of pure ice

Usage example:

```
from smrt.permittivity.ice import ice_permittivity_maetzler87
eps_ice = ice_permittivity_maetzler87(frequency=18e9, temperature=270)
```

Note: This is only suitable for testing at -5 deg C and -15 deg C. If used at other temperatures a warning will be displayed.

---

ice_permittivity_tiuri84 (*frequency*, *temperature*)

Calculates the complex ice dielectric constant depending on the frequency and temperature

Based on Tiuri et al. (1984). The Complex Dielectric Constant of Snow at Microwave Frequencies. IEEE Journal of Oceanic Engineering, vol. 9, no. 5., pp. 377-382

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

Returns Complex permittivity of pure ice

Usage example:

```
from smrt.permittivity.ice import ice_permittivity_tiuri84
eps_ice = ice_permittivity_tiuri84(frequency=1.9e9, temperature=250)
```

2.5 smrt.permittivity.saline_ice module

impure_ice_permittivity_maetzler06 (*frequency*, *temperature*, *salinity*)

Computes permittivity of impure ice from Maetzler 2006 - Thermal Microwave Radiation: Applications for Remote Sensi

Model developed for salinity around 0.013 PSU. The extrapolation is based on linear assumption to salinity, so it is not recommended for much higher salinity.

param temperature ice temperature in K

param salinity salinity of ice in kg/kg (see PSU constant in smrt module)

Usage example:

```
from smrt.permittivity.saline_ice import impure_ice_permittivity_maetzler06
eps_ice = impure_ice_permittivity_maetzler06(frequency=18e9, temperature=270,␣
↪salinity=0.013)
```

saline_ice_permittivity_pvs_mixing(*frequency*, *temperature*, *brine_volume_fraction*, *brine_inclusion_shape='spheres'*, *brine_mixing_ratio=1*, *ice_permittivity_model=None*, *brine_permittivity_model=None*)

Computes effective permittivity of saline ice using the Polder Van Santen mixing formulaes.

Parameters

- **frequency** – frequency in Hz
- **temperature** – ice temperature in K
- **brine_volume_fraction** – brine / liquid water fraction in sea ice
- **brine_inclusion_shape** – Assumption for shape(s) of brine inclusions. Can be a string for single shape, or a list/tuple/dict of strings for mixture of shapes. So far, we have the following shapes: "spheres" and "random_needles" (i.e. randomly-oriented elongated ellipsoidal inclusions). If the argument is a dict, the keys are the shapes and the values are the mixing ratio. If it is a list, the mixing_ratio argument is required.
- **brine_mixing_ratio** – The mixing ratio of the shapes. This is only relevant when inclusion_shape is a list/tuple. Mixing ratio must be a sequence with length len(inclusion_shape)-1. The mixing ratio of the last shapes is deduced as the sum of the ratios must equal to 1.
- **ice_permittivity_model** – pure ice permittivity formulation (default is ice_permittivity_matzler87)
- **brine_permittivity_model** – brine permittivity formulation (default is brine_permittivity_stogryn85)

2.6 smrt.permittivity.saline_snow module

saline_snow_permittivity_geldsetzer09(*frequency*, *density*, *temperature*, *salinity*)

Computes permittivity of saline snow using the frequency dispersion model published by Geldsetzer et al., 2009 (CRST). DOI: 10.1016/j.coldregions.2009.03.009. In-situ measurements collected had salinity concentration between 0.1e-3 and 12e3 kg/kg, temperatures ranging between 257 and 273 K, and a mean snow density of 352 kg/m3.

Validity between 10 MHz and 40 GHz.

Source: Matlab code, Ludovic Brucker

Parameters

- **frequency** – frequency in Hz
- **density** – snow density in kg m-3
- **temperature** – ice temperature in K
- **salinity** – salinity of ice in kg/kg (see PSU constant in smrt module)

saline_snow_permittivity_scharien_with_stogryn71(*frequency*, *density*, *temperature*, *salinity*)

Computes permittivity of saline snow. See *saline_snow_permittivity_scharien* documentation

saline_snow_permittivity_scharien_with_stogryn95 (*frequency*, *density*, *temperature*, *salinity*)
Computes permittivity of saline snow. See *saline_snow_permittivity_scharien* documentation

saline_snow_permittivity_scharien (*density*, *temperature*, *salinity*, *brine_permittivity*)
Computes permittivity of saline snow using the Denoth / Matzler Mixture Model - Dielectric Contsant of Saline Snow.

Assumptions: (1) Brine inclusion geometry as oblate spheroids

Depolarization factor, A0 = 0.053 (Denoth, 1980)

(2) Brine inclusions are isotropically oriented Coupling factor, X = 2/3 (Drinkwater and Crocker, 1988)

Validity ranges:

(1) Temperature, Ts, down to - 22.9 degrees Celcius;

(2) Brine salinity, Sb, up to 157ppt; i.e.up to a Normality of 3 for NaCl Not valid for wet snow

Source: Matlab code, Randall Scharien

Parameters

- **density** – snow density in kg m-3
- **temperature** – snow temperature in K
- **salinity** – snow salinity in kg/kg (see PSU constant in smrt module)
- **brine_permittivity** – brine_permittivity

2.7 smrt.permittivity.saline_water module

seawater_permittivity_klein76 (*frequency*, *temperature*, *salinity*)
Calculates permittivity (dielectric constant) of water using an empirical relationship described by Klein and Swift (1976).

Parameters

- **frequency** – frequency in Hz
- **temperature** – water temperature in K
- **salinity** – water salinity in kg/kg (see PSU constant in smrt module)

Returns complex water permittivity for a frequency f.

seawater_permittivity_stogryn71 (*frequency*, *temperature*)
Computes dielectric constant of brine, complex_b (Stogryn, 1971 approach)

Input parameters: from polynomial fit in Stogryn and Desargent, 1985

Source: Matlab code, Ludovic Brucker

Parameters

- **frequency** – frequency in Hz
- **temperature** – water temperature in K

Returns complex water permittivity for a frequency f.

brine_permittivity_stogryn85 (*frequency*, *temperature*)
computes permittivity and loss of brine using equations given in Stogryn and Desargant (1985): 'The Dielectric Properties of Brine in Sea Ice at Microwave Frequencies', IEEE.

Parameters

- **frequency** – em frequency [Hz]
- **temperature** – ice temperature in K

seawater_permittivity_stogryn95 (*frequency*, *temperature*, *salinity*)
Computes seawater dielectric constant using Stogryn 1995.

source: Stogryn 1995 + http://rime.aos.wisc.edu/MW/models/src/eps_sea_stogryn.f90; Matlab code, Ludovic Brucker

Parameters

- **frequency** – frequency in Hz
- **temperature** – water temperature in K
- **salinity** – water salinity in kg/kg (see PSU constant in smrt module)

Returns complex water permittivity for a frequency f.

2.8 smrt.permittivity.snow_mixing_formula module

Mixing formulae relevant to snow. This module contains equations to compute the effective permittivity of snow.

Note that by default most emmodels (IBA, DMRT, SFT Rayleigh) uses the generic mixing formula Polder van Staten that mixes the permittivities of the background (e.g.) and the scatterer materials (e.g. ice) to compute the effective permittivity of snow in a proportion determined by frac_volume. See py:meth:*~smrt.emmolde.derived_IBA*.

Many semi-empirical mixing formulae have been developed for specific mixture of materials (e.g. snow). They can be used to replace the Polder van Staten in the EM models. They should not be used to set the material permittivities as input of py:meth:*~smrt.smrt_inputs.make_snowpack* and similar functions (because the emmodel would re-mix the already mixed materials with the background material).

wetsnow_permittivity_tinga73 (*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity proposed by Tinga et al. 1973 for three-component mixing. The component 1 is the background ("a"
the compoment 2 ("w" here) is a spherical shell surrounding the component 3 ("i" here).

It was used by Tiuri as well as T. Mote to compute wet snolw permittivity.

Tinga, W.R., Voss, W.A.G. and Blossey, D. F.: General approach to multiphase dielectric mixture theory. Journal of Applied Physics, Vol.44(1973) No.9,pp.3897-3902. doi: /10.1063/1.1662868

Tiuri, M. and Schultz, H., Theoretical and experimental studies of microwave radiation from a natural snow field. In Rango, A. , ed. Microwave remote sensing of snowpack properties. Proceedings of a workshop … Fort Collins, Colorado, May 20-22, 1980. Washington, DC, National Aeronautics and Space Center, 225-234. (Conference Publication 2153.)

compute_frac_volumes (*density*, *liquid_water*)
compute the fractional volume of ice+water, the fractional volume of ice, and the fractional volume of water from the (wet) snow density and the liquid_water which is the volume fraction of liquid with respect to ice + liquid (but no air).

Parameters

- **density** – density of the snow, including the ice and water phases.
- **liquid_water** – (fractional volume of water with respect to ice+water volume).

Returns frac_volume, fi, fw

wetsnow_permittivity_colbeck80_caseI(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity proposed by Colbeck, 1980 for the pendular regime.

Colbeck, S. C. (1980). Liquid distribution and the dielectric constant of wet snow. Goddard Space Flight Center Microwave Remote Sensing of Snowpack Properties, 21–40.

wetsnow_permittivity_colbeck80_caseII(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity proposed by Colbeck, 1980 for the funicular regime and low dry snow density.

Colbeck, S. C. (1980). Liquid distribution and the dielectric constant of wet snow. Goddard Space Flight Center Microwave Remote Sensing of Snowpack Properties, 21–40.

wetsnow_permittivity_colbeck80_caseIII(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity proposed by Colbeck, 1980 for the low porosity.

Colbeck, S. C. (1980). Liquid distribution and the dielectric constant of wet snow. Goddard Space Flight Center Microwave Remote Sensing of Snowpack Properties, 21–40.

wetsnow_permittivity_hallikainen86(*frequency*, *density*, *liquid_water*)

effective permittivity of a snow mixture calculated with the Modified Debye model by Hallikainen 1986

The implemented equation are 10, 11 and 13a-c.

The validity of the model is: frequency between 3 and 37GHz; mv between 1% and 12%; dry_snow_density between 0.09 and 0.38g/cm3.

The implementation of this function follows the equations formulation of the original paper Hallikainen, M., F. Ulaby, and M. Abdelrazik, "Dielectric properties of snow in 3 to 37 GHz range," IEEE Trans. on Antennasand Propagation,Vol. 34, No. 11, 1329–1340, 1986. DOI: 10.1109/TAP.1986.1143757 Anyway this formulation does not allow the reproduction of the results as reported in the paper. A new formulation of eq. 12a have been presented in the book Microwave Radar and Radiometric Remote Sensing by Ulaby et al. 2014 from which the SMRT function wetsnow_permittivity_hallikainen86_ulaby14 have been implemented. The users are pointed to that definition.

wetsnow_permittivity_hallikainen86_ulaby14(*frequency*, *density*, *liquid_water*)

effective permittivity of a snow mixture calculated with the Modified Debye model by Hallikainen 1986 and revised in Microwave Radar and Radiometric Remote Sensing by Ulaby et al. 2014 Equations implemented are ch 4 pp 143-15 4.60a - 4.61h.

The validity of the model is: frequency between 3 and 37GHz; mv between 1% and 12%; dry_snow_density between 0.09 and 0.38g/cm3.

Same formulation can be reproduced by the book code https://mrs.eecs.umich.edu/codes/Module4_6/Module4_6.html

wetsnow_permittivity_wiesmann99(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*)

effective permittivity of a snow mixture as presented in MEMLS by Wiesmann and Matzler, 1999. Note that the version implemented in MEMLS v3 is different.

wetsnow_permittivity_memls(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity of a snow mixture as calculated in MEMLS using Maxwell-Garnett Mixing rule of water in dry snow for prolate spheroidal water with experimentally determined. Dry snow permittivity is here determined with Polder van Santen.

wetsnow_permittivity_three_component_polder_van_santen(*frequency*, *temperature*, *density*, *liquid_water*, *ice_permittivity_model=None*, *water_permittivity_model=None*)

effective permittivity of a snow mixture using the three components polder_van_santen, assuming spherical inclusions

depolarization_factors_maetzler96(*density*)

The empirical depolarization factors of snow estimated by Mäzler 1996. It is supposed to provide more accurate permittivity=f(density) than using constant depolarization factors in Polder van Santen (e.g. spheres)

Biblio: C. Mäzler, Microwave Permittivity of dry snow, IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, VOL. 34, NO. 2, MARCH 1996

drysnow_permittivity_maetzler96(*density*, *e0=1*, *eps=3.185*)

default_ice_water_permittivity(*ice_permittivity_model*, *water_permittivity_model*)

2.9 smrt.permittivity.test_generic_mixing_formula module

test_isotropic_default_depolarization_factors()

test_plates_depol()

test_hoar_columns_depol()

test_depol_approach_to_isotropy_above()

test_depol_approach_to_isotropy_below()

test_pvsl_spheres()

test_pvsl_needles()

test_pvsl_mix_spheres_needles()

2.10 smrt.permittivity.test_ice module

test_ice_permittivity_output_matzler_temp_270()

test_ice_permittivity_output_matzler_temp_250()

test_imaginary_ice_permittivity_output_matzler_temp_270_freq_20GHz()

test_imaginary_ice_permittivity_output_matzler_temp_250_freq_20GHz()

test_imaginary_ice_permittivity_output_matzler_temp_270_freq_30GHz()

test_imaginary_ice_permittivity_output_matzler_temp_250_freq_30GHz()

test_imaginary_ice_permittivity_output_matzler_temp_270_freq_40GHz()

test_imaginary_ice_permittivity_output_matzler_temp_250_freq_40GHz()

test_real_ice_permittivity_output_maetzler87_temp_268()

test_imag_ice_permittivity_output_maetzler87_temp_minus5()

test_imag_ice_permittivity_output_maetzler87_temp_minus15()

test_ice_permittivity_output_tuiri84_temp_minus10_freq_10GHz()

test_ice_permittivity_output_tuiri84_temp_minus10_freq_40GHz()

test_ice_permittivity_output_tuiri84_temp_250K_freq_10GHz()

test_ice_permittivity_output_tuiri84_temp_250K_freq_40GHz()

test_real_ice_permittivity_output_HUT()

test_imaginary_ice_permittivity_output_HUT()

test_real_ice_permittivity_output_DMRTML()

test_imaginary_ice_permittivity_output_DMRTML()

test_real_ice_permittivity_output_matzler_temp_270_MEMLS()

test_imaginary_ice_permittivity_output_matzler_temp_270_freq_10GHz()

test_salty_imaginary_ice_permittivity_output_matzler_temp_270_freq_10GHz()

2.11 smrt.permittivity.test_saline_ice module

test_impure_permittivity_same_as_pure_for_zero_salinty()

test_impure_ice_freezing_point_0p013psu_10GHz()

test_saline_permittivity_same_as_pure_for_zero_salinity()

test_saline_permittivity_with_mixtures()

2.12 smrt.permittivity.water module

water_permittivity_maetzler87(*frequency*, *temperature*)

Calculates the complex water dielectric constant depending on the frequency and temperature Based on Mätzler, C., & Wegmuller, U. (1987). Dielectric properties of freshwater ice at microwave frequencies. *Journal of Physics D: Applied Physics*, 20(12), 1623-1630.

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

Raises **Exception** – if liquid water > 0 or salinity > 0 (model unsuitable)

Returns Complex permittivity of pure ice

water_permittivity(*frequency*, *temperature*)

Calculates the complex water dielectric constant depending on the frequency and temperature Based on Mätzler, C., & Wegmuller, U. (1987). Dielectric properties of freshwater ice at microwave frequencies. *Journal of Physics D: Applied Physics*, 20(12), 1623-1630.

Parameters

- **frequency** – frequency in Hz

- **temperature** – temperature in K

Raises **Exception** – if liquid water > 0 or salinity > 0 (model unsuitable)

Returns Complex permittivity of pure ice

water_permittivity_tiuri80 (*frequency*, *temperature*)

Calculates the complex water dielectric constant reported by:

Tiuri, M. and Schultz, H., Theoretical and experimental studies of microwave radiation from a natural snow field. In Rango, A. , ed.

Microwave remote sensing of snowpack properties. Proceedings of a workshop ... Fort Collins, Colorado, May 20-22, 1980. Washington, DC, National Aeronautics and Space Center, 225-234. (Conference Publication 2153.)

https://ntrs.nasa.gov/api/citations/19810010984/downloads/19810010984.pdf

2.13 smrt.permittivity.wetice module

wetice_permittivity_bohren83 (*frequency*, *temperature*, *liquid_water*)

calculate the dielectric constant of wet particules of ice using Maxwell Garnet equation using water as the background and ice as the inclusions. As reported by Bohren and Huffman 1983 according to Ya Qi Jin, eq 8-69, 1996 p282

see also: K L CHOPRA and G B REDDY, Praman.a- Optically selective coatings, J. Phys., Vol. 27, Nos 1 & 2, July & August 1986, pp. 193-217.

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

:param liquid_water (fractional volume of water with respect to ice+water volume). :returns: Complex permittivity of pure ice

symmetric_wetice_permittivity (*frequency*, *temperature*, *liquid_water*)

calculate the dielectric constant of wet particules of ice using Polder van Santen Maxwell equation assuming both ice and water are fully mixed. This applies to intermediate content of wetness. Typically liquid_water=0.5.

Parameters

- **frequency** – frequency in Hz
- **temperature** – temperature in K

:param liquid_water (fractional volume of water with respect to ice+water volume). :returns: Complex permittivity of pure ice

2.14 smrt.permittivity.wetsnow module

wetsnow_permittivity (*frequency*, *temperature*, *liquid_water*)

calculate the dielectric constant of wet particule of ice using Bohren and Huffman 1983 according to Ya Qi Jin, eq 8-69, 1996 p282

Parameters

- **frequency** – frequency in Hz

- **temperature** – temperature in K

:param liquid_water (fractional volume of water with respect to ice+water volume) :returns: Complex permittivity of pure ice

2.15 Module contents

This module contains permittivity formulations for different materials. They are organised in different files for easy access but this is not strictly required.

E.g. ice.py contains formulation for pure ice permittivity.

For developers

To add a new permittivity function proceed as follows:

1. To add a new permittivity formulation add a function either in an existing file or in a new file (recommended for testing). E.g. for salty ice permittivity formulations should be in saltyice.py and so on.

2. Any function defining a permittivity model must declare the mapping between the layer properties and the arguments of the function (see ice.py for examples). It means that the arguments of the function must be listed (in order) in the @required_layer_properties decorator. In most cases, the name of the arguments should be the same as a properties, but this is not strictly necessary, only the order matters. E.g.:

```
@required_layer_properties("temperature", "salinity")
def permittivity_something(frequency, t, s):
```

maps the layer property "temperature" to the argument "t" of the function (and "salinity" to s) However, we recommend to change t into temperature for sake of clarity.

For curious ones, this declaration is required because the function can be called either with its arguments (normal case) or with only two arguments like this (frequency, layer). In this latter case, the arguments required by the original function are automatically extracted from the layer attributes (=properties) based on the declaration in @required_layer_properties. This complication is necessary because there is no way in Python to inspect the name of the arguments of a function, so the need for explicit declaration.

3. to use the new function, import the module (e.g. from smrt.permittivity.ice import permittivity_something) and pass this function to `smrt.core.snowpack.make_snowpack()` or `smrt.core.layer:make_snow_layer()`.

CHAPTER 3

smrt.microstructure_model package

3.1 Submodules

3.2 smrt.microstructure_model.autocorrelation module

This module contains the base classes for the microstructure classes. **It is not used directly**.

class AutocorrelationBase(*params*)
Bases: `object`

Low level base class for the Autocorrelation base class to handle optional and required arguments. **It should not be used directly**.

classmethod compute_all_arguments()

classmethod valid_arguments()

class Autocorrelation(*params*)
Bases: *smrt.microstructure_model.autocorrelation.AutocorrelationBase*

Base class for autocorrelation function classes. It should not be used directly but sub-classed. It provides generic handling of the numerical fft and invfft when required by the user or when necessary due to the lack of implementation of the real or ft autocorrelation functions. See the source of *Exponential* to see how to use this class.

args = []

optional_args = {'ft_numerical': False, 'real_numerical': False}

ft_autocorrelation_function_fft(*k*)
compute the fourier transform of the autocorrelation function via fft Args: k: array of wave vector magnitude values, ordered, and non-negative

autocorrelation_function_invfft(*r*)
Compute the autocorrelation function from an analytically known FT via fft Args: r: array of lag vector magnitude values, ordered, non-negative

inverted_medium()
return the same autocorrelation for the inverted medium. In general, it is only necessary to invert the fractional volume if the autocorrelation function is numerically symmetric as it should be. This needs to be reimplemented in the sub classes if this is not sufficient.

3.3 smrt.microstructure_model.exponential module

Exponential autocorrelation function model of the microstructure. This microstructure model is used by MEMLS when IBA is selected.

parameters: frac_volume, corr_length

class Exponential(*params*)
Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'corr_length']

optional_args = {}

corr_func_at_origin

inv_slope_at_origin

basic_check()
check consistency between the parameters

compute_ssa()
compute the ssa for the exponential model according to Debye 1957. See also Maetzler 2002 Eq. 11

autocorrelation_function(*r*)
compute the real space autocorrelation function

ft_autocorrelation_function(*k*)
compute the fourier transform of the autocorrelation function analytically

3.4 smrt.microstructure_model.gaussian_random_field module

Gaussian Random field model of the microstructure.

parameters: frac_volume, corr_length, repeat_distance

class GaussianRandomField(*params*)
Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'corr_length', 'repeat_distance']

optional_args = {}

corr_func_at_origin

inv_slope_at_origin

basic_check()
check consistency between the parameters

compute_ssa()
Compute the ssa for a sphere

autocorrelation_function(*r*)
compute the real space autocorrelation function for the Gaussian random field model

3.5 smrt.microstructure_model.homogeneous module

Homogeneous microstructure. This microstructure model is to be used with non-scattering emmodel.

parameters: none

class Homogeneous(*params*)

Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume']

optional_args = {}

corr_func_at_origin

inv_slope_at_origin

basic_check()
check consistency between the parameters

compute_ssa()
compute the ssa of an homogeneous medium

autocorrelation_function(*r*)
compute the real space autocorrelation function

ft_autocorrelation_function(*k*)
compute the fourier transform of the autocorrelation function analytically

3.6 smrt.microstructure_model.independent_sphere module

Independent sphere model of the microstructure.

parameters: frac_volume, radius

class IndependentSphere(*params*)

Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'radius']

optional_args = {}

corr_func_at_origin

inv_slope_at_origin

basic_check()
check consistency between the parameters

compute_ssa()
Compute the ssa for a sphere

autocorrelation_function(*r*)
compute the real space autocorrelation function for an independent sphere

ft_autocorrelation_function(*k*)
Compute the 3D Fourier transform of the isotropic correlation function for an independent sphere for given magnitude k of the 3D wave vector (float).

3.7 smrt.microstructure_model.sampled_autocorrelation module

Sampled autocorrelation function model. To use when no analytical form of the autocorrelation function but the values of the autocorrelation function (*acf*) is known at a series of *lag*.

parameters: frac_volume, lag, acf

acf contains the values at different *lag*. These parameters must be lists or arrays.

class SampledAutocorrelation(*params*)

Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'lag', 'acf']

optional_args = {}

corr_func_at_origin

basic_check()

check consistency between the parameters

compute_ssa()

compute the ssa according to Debye 1957. See also Maetzler 2002 Eq. 11

autocorrelation_function(*r*)

compute the real space autocorrelation function by interpolation of requested values from known values

3.8 smrt.microstructure_model.sticky_hard_spheres module

Monodisperse sticky hard sphere model of the microstructure.

parameters: frac_volume, radius, stickiness.

The stickiness is optional but it is recommended to use value around 0.2 as a first guess. Be aware that low values of stickiness are invalid, the limit depends on the fractional volume (see for instance Loewe and Picard, 2015). See the *tau_min()* method.

Currently the implementation is specific to ice / snow. It can not be used for other materials.

class StickyHardSpheres(*params*)

Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'radius']

optional_args = {'stickiness': 1000}

corr_func_at_origin

inv_slope_at_origin

basic_check()

check consistency between the parameters

compute_ssa()

Compute the ssa of a sphere assembly

ft_autocorrelation_function(*k*)

Compute the 3D Fourier transform of the isotropic correlation function for sticky hard spheres in Percus–Yevick approximation for given magnitude k of the 3D wave vector (float).

compute_t()

compute the t parameter used in the stickiness

tau_min(*frac_volume*)
compute the minimum possible stickiness value for given ice volume fraction

3.9 smrt.microstructure_model.test_autocorrelation module

3.10 smrt.microstructure_model.test_exponential module

test_constructor()

3.11 smrt.microstructure_model.test_sticky_hard_spheres module

test_constructor()

test_autocorrelation()

3.12 smrt.microstructure_model.teubner_strey module

Teubner Strey model model of the microstructure.

parameters: frac_volume, corr_length, repeat_distance

class TeubnerStrey(*params*)
Bases: *smrt.microstructure_model.autocorrelation.Autocorrelation*

args = ['frac_volume', 'corr_length', 'repeat_distance']

optional_args = {}

corr_func_at_origin

basic_check()
check consistency between the parameters

compute_ssa()
Compute the ssa for a sphere

autocorrelation_function(*r*)
compute the real space autocorrelation function for the Teubner Strey model

ft_autocorrelation_function(*k*)
Compute the 3D Fourier transform of the isotropic correlation function for Teubner Strey for given magnitude k of the 3D wave vector (float).

3.13 Module contents

Microstructure models are different representations of the snow microstructure. Because these representations are different, the parameters to describe actual snow micro-structure depends on the model. For instance, the Sticky Hard Spheres medium is implemented in *sticky_hard_spheres* and its parameters are: the *radius* (required) and the *stickiness* (optional, default value is non-sticky, even though we do recommend to use a stickiness of ~0.1-0.3 in practice).

Because IBA is one of the important electromagnetic theories provided by SMRT, the first/main role of microstructure models is to provide the Fourier transform of the autocorrelation functions. Hence most microstructure models are named after the autocorrelation function. For instance, the *exponential* autocorrelation function is that used in MEMLS. Its only parameter is the *corr_length*.

To use microstructure models, it is only required to read the documentation of each model to determine the required and optional parameters. Selecting the microstructure model is usually done with make_snowpack which only requires the name of the module (the filename with .py). The import of the module is automatic. For instance:

```
from smrt import make_snowpack

sp = make_snowpack([1, 1000], "exponential", density=[200, 300], corr_length=[0.2e-3,␣
↪0.5e-3])
```

This snippet creates a snowpack with the exponential autocorrelation function for all (2) layers. Import of the *exponential* is automatic and creation of instance of the class *Exponential* is done by the model *smrt.core.model.Model.run()* method.

CHAPTER 4

smrt.interface package

4.1 Submodules

4.2 smrt.interface.coherent_flat module

Implement the coherent flat pseudo-interface, as in MEMLS. This interface is obtained by collapsing one layer and two interfaces into a single interface. Scattering in the layer is neglected.

process_coherent_layers(*snowpack*, *emmodel_list*, *sensor*)

class CoherentFlat(*interfaces*, *layer*, *permittivity*)

Bases: object

A flat surface. The reflection is in the specular direction and the coefficient is calculated with the Fresnel coefficients

args = []

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium.
- **mu1** – array of cosine of incident angles.
- **npol** – number of polarization.

Returns the reflection matrix

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for the azimuthal mode m and for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium.
- **mu1** – array of cosine of incident angles.
- **npol** – number of polarization.

Returns the transmission matrix

diffuse_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

4.3 smrt.interface.flat module

Implement the flat interface boundary condition between layers charcterized by their effective permittivities. The reflection and transmission are computed using the Fresnel coefficient.

class Flat(***kwargs*)

Bases: *smrt.core.interface.Interface*

A flat surface. The reflection is in the specular direction and the coefficient is calculated with the Fresnel coefficients

args = []

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium.
- **mu1** – array of cosine of incident angles.
- **npol** – number of polarization.

Returns the reflection matrix

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.

- **eps_2** – permittivity of the other medium.
- **mu1** – array of cosine of incident angles.
- **npol** – number of polarization.

Returns the transmission matrix

diffuse_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

4.4 smrt.interface.geometrical_optics module

Implement the interface boundary condition under the Geometrical Approximation between layers charcterized by their effective permittivities. This approximation is suitable for surface with roughness much larger than the roughness scales, typically k*s >> 1 and k*l >> 1, where s the rms heigth and l the correlation length. The precise validity range must be investigated by the user, this code does not raise any warning. An important charcateristic of this approximation is that the scattering do not directly depend on frequency, the only (probably weak) dependence is through the permittivities of the media.

The model is parameterized by the mean_square_slope which can be calculated as mean_square_slope = 2*s**2/l**2 for surface with a Gaussian autocorrelation function. Other equations may exist for other autocorrelation function.

This implementation is largely based on Tsang and Kong, Scattering of Electromagnetic Waves: Advanced Topics, 2001 (Tsang_tomeIII in the following)

class GeometricalOptics(***kwargs*)

Bases: *smrt.core.interface.Interface*

A very rough surface.

args = ['mean_square_slope']

optional_args = {'shadow_correction': True}

clip_mu(*mu*)

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

compute the reflection coefficients for an array of incident, scattered and azimuth angles in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.

- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *m_max*, *npol*)

ft_even_diffuse_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *m_max*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for the azimuthal mode m and for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the transmission matrix

diffuse_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu_t*, *mu_i*, *dphi*, *npol*)

compute the transmission coefficients for an array of incident, scattered and azimuth angles in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu_i** – array of cosine of incident angles
- **mu_t** – array of cosine of transmitted wave angles
- **npol** – number of polarization

Returns the transmission matrix

reflection_integrand_for_energy_conservation_test(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*)

function relevant to compute energy conservation. See p87 in Tsang_tomeIII.

transmission_integrand_for_energy_conservation_test(*frequency*, *eps_1*, *eps_2*, *mu_t*, *mu_i*, *dphi*)

function relevant to compute energy conservation. See p87 in Tsang_tomeIII.

reflection_coefficients(*frequency*, *eps_1*, *eps_2*, *mu_i*)

transmission_coefficients(*frequency*, *eps_1*, *eps_2*, *mu_i*)

shadow_function(*mean_square_slope*, *cotan*)

4.5 smrt.interface.geometrical_optics_backscatter module

Implement the interface boundary condition under the Geometrical Approximation between layers charcterized by their effective permittivities. This code is for backscatter only, that is, to use as a substrate and at low frequency when the backscatter is the main mecahnism, and conversely when mulitple scattering and double bounce between snow and substrate are negligible. In any other case, it is recommended to use *GeometricalOptics*.

The transmitted energy is also computed in an approximate way suitable for 1st order scattering. We use energy conservation to compute the total transmitted energy and consider that all this energy is transmitted in the refraction (specular) direction.

class GeometricalOpticsBackscatter(***kwargs*)

Bases: *smrt.core.interface.Interface*

A very rough surface.

args = ['mean_square_slope']

optional_args = {'shadow_correction': True}

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

compute the reflection coefficients for an array of incident, scattered and azimuth angles in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *m_max*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted. While Geometrical Optics, we here consider that power not reflected is scattered in the specular transmitted direction. This is an approximation which is reasonable in the context of a "1st order" geometrical optics.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the transmission matrix

4.6 smrt.interface.iem_fung92 module

Implement the interface boundary condition under IEM formulation provided by Fung et al. 1992. in IEEE TGRS. Use of this code requires special attention because of two issues:

1) it only computes the backscatter diffuse reflection as described in Fung et al. 1992, the specular reflection and the coherent transmission. It does not compute the full bi-static diffuse reflection and transmission. As a consequence it is only suitable when single scattering is dominant, usually at low frequency when the medium is weakly scattering. This happends when the main mechanism is the backscatter from the interface attenuated by the medium. Another case is when the rough surface is relatively smooth, the specular reflection is relatively strong and the energy can be scattered back by the medium (double bounce). For other situations, this code is not recommended.

2) Additionaly, IEM is known to work for a limited range of roughness. Usually it is considered valid for ks < 3 and ks*kw < sqrt(eps) where k is the wavenumber, s the rms height and l the correlation length. The code print a warning when out of this range. There is also limitation for smooth surfaces but no warning is printed.

Usage example:

:: # rms height and corr_length values work at 10 GHz substrate = make_soil("iem_fung92", "dobson85", temperature=260,

roughness_rms=1e-3, corr_length=5e-2, autocorrelation_function="exponential", moisture=moisture, clay=clay, sand=sand, drymatter=drymatter)

class IEM_Fung92 (***kwargs*)

Bases: *smrt.core.interface.Interface*

A moderate rough surface model with backscatter, specular reflection and transmission only. Use with care!

args = ['roughness_rms', 'corr_length']

optional_args = {'autocorrelation_function': 'exponential', 'series_truncation': 10,

specular_reflection_matrix (*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

check_validity(*ks*, *kl*, *eps_r*)

fresnel_coefficients(*eps_1*, *eps_2*, *mu_i*, *ks*, *kl*)
calculate the fresnel coefficients at the angle mu_i whatever is ks and kl according to the original formulation of Fung 1992

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*, *debug=False*)

compute the reflection coefficients for an array of incident, scattered and azimuth angles in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

W_n(*n*, *k*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *m_max*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for the azimuthal mode m and for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the transmission matrix

4.7 smrt.interface.iem_fung92_brogioni10 module

Implement the interface boundary condition under IEM formulation provided by Fung et al. 1992. in IEEE TGRS with an extended domain of validity (for large roughness or correlation length) by switching the Fresnel coefficients according to Brogioni et al. 2010, IJRS (doi: 10.1080/01431160903232808). A better but more complex approach is given by Wu et al. 2003 (to be implemented).

Use of this code requires special attention.

class IEM_Fung92_Briogoni10(***kwargs*)
Bases: *smrt.interface.iem_fung92.IEM_Fung92*

A moderate rough surface model with backscatter, specular reflection and transmission only. Use with care!

check_validity(*ks*, *kl*, *eps_r*)

fresnel_coefficients(*eps_1*, *eps_2*, *mu_i*, *ks*, *kl*)
calculate the fresnel coefficients at the angle mu_i or 0° depending on ks*kl. The transition is abrupt.

4.8 smrt.interface.radar_calibration_sphere module

Surface with a backscatter of 4pi.

class RadarCalibrationSphere(***kwargs*)

Bases: *smrt.core.interface.Interface*

args = []

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *m_max*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

4.9 smrt.interface.test_geometrical_optics module

test_compare_geometrical_optics()

test_reflectance_reciprocity()

test_transmission_reciprocity()

4.10 smrt.interface.test_iem_fung92 module

test_iem_fung92()

4.11 smrt.interface.test_iem_fung92_brogioni10 module

test_iem_fung92_biogoni10_continuty()

test_iem_fung92_brogioni10()

4.12 smrt.interface.transparent module

A transparent interface (no reflection). Useful for the unit-test mainly.

class Transparent

Bases: object

args = []

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the reflection coefficients for the azimuthal mode m and for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mhu1** – array of cosine of incident angles
- **npol** – number of polarization

diffuse_reflection_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

coherent_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu1*, *npol*)

compute the transmission coefficients for the azimuthal mode m and for an array of incidence angles (given by their cosine) in medium 1. Medium 2 is where the beam is transmitted.

Parameters

- **eps_1** – permittivity of the medium where the incident beam is propagating.
- **eps_2** – permittivity of the other medium
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the transmission matrix

diffuse_transmission_matrix(*frequency*, *eps_1*, *eps_2*, *mu_s*, *mu_i*, *dphi*, *npol*)

4.13 smrt.interface.vector3 module

class vector3

Bases: object

static from_xyz(*x*, *y*, *z*)

static from_angles(*norm*, *mu*, *phi*)

cross(*other*)

dot(*other*)

4.14 Module contents

This module contains different type of boundary conditions between the layers. Currently only flat interfaces are implemented.

For developers

All the different type of interface must defined the methods: *specular_reflection_matrix* and *coherent_transmission_matrix*.

It is currently not possible to implement rough interface, a (small) change is needed in DORT. Please contact the authors.

CHAPTER 5

smrt.substrate package

5.1 Submodules

5.2 smrt.substrate.flat module

Implement the flat interface boundary for the bottom layer (substrate). The reflection and transmission are computed using the Fresnel coefficients. This model does not take any specific parameter.

class Flat (*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = []

diffuse_reflection_matrix (*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

optional_args = {}

specular_reflection_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

5.3 smrt.substrate.geometrical_optics module

Implement the geometrical optics rough substrate. See the documentation in *geometrical_optics*

class GeometricalOptics (*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = ['mean_square_slope']

diffuse_reflection_matrix (*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix (*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

optional_args = {'shadow_correction': True}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.4 smrt.substrate.geometrical_optics_backscatter module

Implement the geometrical optics rough substrate. See the documentation in *geometrical_optics_backscatter*.

class GeometricalOpticsBackscatter(*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = ['mean_square_slope']

diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

optional_args = {'shadow_correction': True}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.5 smrt.substrate.iem_fung92 module

Implement the flat interface boundary for the bottom layer (substrate). The reflection and transmission are computed using the Fresnel coefficients. This model does not take any specific parameter.

class IEM_Fung92(*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = ['roughness_rms', 'corr_length']

diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

optional_args = {'autocorrelation_function': 'exponential', 'series_truncation': 10,

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.6 smrt.substrate.iem_fung92_brogioni10 module

Implement the flat interface boundary for the bottom layer (substrate). The reflection and transmission are computed using the Fresnel coefficients. This model does not take any specific parameter.

class IEM_Fung92_Briogoni10(*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = ['roughness_rms', 'corr_length']

diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

optional_args = {'autocorrelation_function': 'exponential', 'series_truncation': 10,

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.7 smrt.substrate.radar_calibration_sphere module

Implement the radar_calibration_sphere interface boundary for the bottom layer (substrate).

class RadarCalibrationSphere(*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = []

diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.8 smrt.substrate.reflector module

Implement a reflective boundary conditions with prescribed reflection coefficient in the specular direction. The reflection is set to a value or a function of theta. Azimuthal symmetry is assumed (no dependence on phi).

The *specular_reflection* parameter can be a scalar, a function or a dictionary.

- scalar: same reflection is use for all angles
- function: the function must take a unique argument theta array (in radians) and return the reflection as an array of the same size as theta
- dictionary: in this case, the keys must be 'H' and 'V' and the values are a scalar or a function and are interpreted as for the non-polarized case.

To make a reflector, it is recommended to use the helper function *make_reflector()*.

Examples:

```
# the full path import is required
from smrt.substrate.reflector import make_reflector

# return a perfect reflector (the temperature is useless in this specific case)
ref = make_reflector(temperature=260, specular_reflection=1)

# return a perfect absorber / black body.
ref = make_reflector(temperature=260, specular_reflection=0)

# Specify a frequency and polarization dictionary of reflectivity
ref = make_reflector(specular_reflection={(21e9, 'H'): 0.5, (21e9, 'V'): 0.6, (36e9,
↪'H'): 0.7, (36e9, 'V'): 0.8})
```

Note: the backscatter coefficient argument is not implemented/documented yet.

make_reflector(*temperature=None*, *specular_reflection=None*)
Construct a reflector or absorber instance.

class Reflector(*temperature=None*, *permittivity_model=None*, ***kwargs*)
Bases: *smrt.core.interface.Substrate*

args = []

optional_args = {'backscatter_coefficient': None, 'specular_reflection': None}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.9 smrt.substrate.reflector_backscatter module

Implement a reflective boundary conditions with prescribed reflection coefficient in the specular direction. The reflection is set to a value or a function of theta. Azimuthal symmetry is assumed (no dependence on phi).

The *specular_reflection* parameter can be a scalar, a function or a dictionary.

- scalar: same reflection is use for all angles
- function: the function must take a unique argument theta array (in radians) and return the reflection as an array of the same size as theta
- dictionary: in this case, the keys must be 'H' and 'V' and the values are a scalar or a function and are interpreted as for the non-polarized case.

To make a reflector, it is recommended to use the helper function *make_reflector()*.

Examples:

```
# the full path import is required
from smrt.substrate.reflector import make_reflector

# return a perfect reflector (the temperature is useless in this specific case)
ref = make_reflector(temperature=260, specular_reflection=1)

# return a perfect absorber / black body.
ref = make_reflector(temperature=260, specular_reflection=0)
```

Note: the backscatter coefficient argument is not implemented/documented yet.

make_reflector(*temperature=None*, *specular_reflection=None*, *backscattering_coefficient=None*)
Construct a reflector or absorber instance.

class Reflector(*temperature=None*, *permittivity_model=None*, ***kwargs*)
Bases: *smrt.core.interface.Substrate*

args = []

optional_args = {'backscattering_coefficient': None, 'specular_reflection': None}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

ft_even_diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *m_max*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.10 smrt.substrate.rough_choudhury79 module

Implement the rough boundary reflectivity presented in Choudhury et al. (1979). It is not suitable for the active mode.

Applicable for ksigma<<1

parameters: roughness_rms

class ChoudhuryReflectivity (*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.Substrate*

args = ['roughness_rms']

optional_args = {}

adjust (*rh*, *rv*, *frequency*, *eps_1*, *mu1*)

specular_reflection_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

emissivity_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

5.11 smrt.substrate.soil_qnh module

Implement the QNH soil model proposed by Wang, 1983. This model is for the passive mode, it is not suitable for the active mode.

parameters: Q, N, H (or Nv and Nh instead of N)

Q and N are set to zero by default. The roughness rms is called H and is a required parameter (note: it is called roughness_rms in soil_wegmuller)

Examples:

soil = make_soil("soil_qnh", "dobson85", moisture=0.2, sand=0.4, clay=0.3, drymatter=1100, Q=0, N=0, H=1e-2)

class SoilQNH (*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.Substrate*

args = ['H']

optional_args = {'N': 0.0, 'Nh': nan, 'Nv': nan, 'Q': 0.0}

adjust (*rh*, *rv*, *mu1*)

specular_reflection_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

emissivity_matrix (*frequency*, *eps_1*, *mu1*, *npol*)

5.12 smrt.substrate.soil_wegmuller module

Implement the empirical soil model presented in Wegmuller and Maetzler 1999. It is often used in microwave radiometry. It is not suitable for the active mode.

parameters: roughness_rms

class SoilWegmuller (*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.Substrate*

args = ['roughness_rms']

optional_args = {}

adjust(*rh*, *rv*, *frequency*, *eps_1*, *mu1*)

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.13 smrt.substrate.test_flat module

test_make_flat()

5.14 smrt.substrate.test_reflector module

test_scalar_specular()

test_dict_specular()

test_func_specular()

test_dict_multifrequency()

test_missing_frequency_warning()

test_emissivity_reflectivity_relation()

test_tuple_dict_multifrequency()

test_inverted_reflector_dictionary()

5.15 smrt.substrate.test_rough_choudhury79 module

test_make_rough_choudhury()

test_rough_choudhury_reflection()

test_raises_ksigma_warning()

test_make_rough_water()

test_equivalence_fresnel()

5.16 smrt.substrate.test_soil_qnh module

test_make_soil_qnh()

test_make_soil_qnh_params()

soil_setup()

test_soil_qnh_reflection()

test_soil_qnh_emissivity()

5.17 smrt.substrate.test_soil_wegmuller module

test_make_soil_wegmuller()

test_soil_wegmuller_reflection()

5.18 smrt.substrate.transparent module

Implement the geometrical optics rough substrate. See the documentation in *geometrical_optics_backscatter*.

class Transparent(*temperature=None*, *permittivity_model=None*, ***kwargs*)

Bases: *smrt.core.interface.SubstrateBase*

args = []

diffuse_reflection_matrix(*frequency*, *eps_1*, *mu_s*, *mu_i*, *dphi*, *npol*)

emissivity_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

optional_args = {}

specular_reflection_matrix(*frequency*, *eps_1*, *mu1*, *npol*)

5.19 Module contents

This directory contains different options to represent the substrate, that is the lower boundary conditions of the radiation transfer equation. This is usually the soil or ice or water but can be an aluminium plate or an absorber.

To create a substrate, use/implement an helper function such as *make_soil()*. This function is able to automatically load a specific soil model .

Examples:

```
from smrt import make_soil
soil = make_soil("soil_wegmuller", "dobson85", moisture=0.2, sand=0.4, clay=0.3,␣
↪drymatter=1100, roughness_rms=1e-2)
```

It is recommended to read first the documentation of *make_soil()* and then explore the different types of soil models.

For developers

To develop a new substrate formulation, you must add a file in the smrt/substrate directory. The name of the file is used by make_soil to build the substrate object.

CHAPTER 6

smrt.atmosphere package

6.1 Submodules

6.2 smrt.atmosphere.simple_isotropic_atmosphere module

Implement an isotropic atmosphere with prescribed frequency-dependent emission (up and down) and transmittivity.

TB and transmissivity can be specified as a constant, or a frequency-dependent dictionary

To make an atmosphere, it is recommended to use the helper function *make_atmosphere()*.

Examples:

```
# the full path import is required
from smrt.atmosphere.simple_isotropic_atmosphere import make_atmosphere

# Constant
atmos = make_atmosphere(tbdown=20., tbup=6., trans=1)

# Frequency-dependent
atmos = make_atmosphere(tbdown={10e9: 15.2, 21e9: 23.5})
```

make_atmosphere (*tbdown=0*, *tbup=0*, *trans=1*)

Construct an atmosphere instance.

class SimpleIsotropicAtmosphere (*tbdown=0*, *tbup=0*, *trans=1*)

Bases: *smrt.core.atmosphere.AtmosphereBase*

tbdown (*frequency*, *costheta*, *npol*)

tbup (*frequency*, *costheta*, *npol*)

trans (*frequency*, *costheta*, *npol*)

6.3 smrt.atmosphere.test_atmosphere module

test_simple_isotropic_atmosphere()

test_frequency_dependent_atmosphere()

6.4 Module contents

This directory contains different options to represent the atmosphere, that is the upper boundary conditions of the radiation transfer equation.

This part is currently not fully developed but should work for an isotropic atmosphere.

Example:

```
from smrt.atmosphere.basic import ConstantAtmosphere

atmosphere = ConstantAtmosphere(tbdown=2.7, tbup=2.7, trans=0.998)
```

The API is subject to change.

CHAPTER 7

smrt.emmodel package

7.1 Submodules

7.2 smrt.emmodel.common module

rayleigh_scattering_matrix_and_angle_tsang00(*mu_s*, *mu_i*, *dphi*, *npol=2*)
compute the Rayleigh matrix and half scattering angle. Based on Tsang theory and application p271 Eq 7.2.16

phase_matrix_from_scattering_amplitude(*fvv*, *fvh*, *fhv*, *fhh*, *npol=2*)
compute the phase function according to the scattering amplitude. This follows Tsang's convention.

extinction_matrix(*sigma_V*, *sigma_H=None*, *npol=2*, *mu=None*)
compute the extinction matrix from the extinction in V and in H-pol. If sigma_V or sigma_H are a scalar, they are expanded in a diagonal matrix provided mu is given. If sigma_H is None, sigma_V is used.

rayleigh_scattering_matrix_and_angle_maetzler06(*mu_s*, *mu_i*, *dphi*, *npol=2*)
compute the Rayleigh matrix and half scattering angle. Based on Mätzler 2006 book p111. This version is relatively slow because it uses phase matrix rotations which is unnecessarily complex for the Rayleigh phase matrix but would be of interest for other phase matrices.

Lmatrix(*cos_phi*, *sin_phi_sign*, *npol*)

rayleigh_scattering_matrix_and_angle(*mu_s*, *mu_i*, *dphi*, *npol=2*)
compute the Rayleigh matrix and half scattering angle. Based on Tsang theory and application p271 Eq 7.2.16

class AdjustableEffectivePermittivityMixins
Bases: `object`

Mixin that allow an EM model to have the effective permittivity model defined by the user instead of by the theory of the EM Model.

The EM model must declare a default effective permittivity model.

effective_permittivity()
Calculation of complex effective permittivity of the medium.

Returns effective_permittivity complex effective permittivity of the medium

derived_EMModel(*base_class*, *effective_permittivity_model*)
return a new IBA model with variant from the default IBA.

Parameters **effective_permittivity_model** – permittivity mixing formula.

:returns a new class inheriting from IBA but with patched methods

7.3 smrt.emmodel.commontest module

test_energy_conservation(*em*, *tolerance_pc*, *npol=None*, *subset=16*)
test energy conservation

Parameters

- **em** – the electromagnetic model that has been set up
- **tolerance_pc** – relative tolerance

7.4 smrt.emmodel.dmrt_qca_shortrange module

Compute scattering with DMRT QCA Short range. Short range means that it is accurate only for small and weakly sticky spheres (high stickiness value). It diverges (increasing scattering coefficient) if these conditions are not met. Numerically the size conditions can be evaluated with the ratio radius/wavelength as for Rayleigh scatterers. For the stickiness, it is more difficult as this depends on the size of the scatterers and the fractional volume. In any case, it is dangerous to use too small a stickiness value, especially if the grains are big.

This model is only compatible with the SHS microstructure model.

Examples:

from smrt import make_snowpack, make_sensor

density = [345.0] temperature = [260.0] thickness = [70] radius = [750e-6] stickiness = [0.1]

snowpack = make_snowpack(thickness, "sticky_hard_spheres", density=density, temperature=temperature, radius=radius, stickiness=stickiness)

create the EM Model - Equivalent DMRTML m = make_model("dmrt_shortrange", "dort")

create the sensor theta = np.arange(5.0, 80.0, 5.0) radiometer = sensor.amsre()

class DMRT_QCA_ShortRange(*sensor*, *layer*, *dense_snow_correction='auto'*)
Bases: *smrt.emmodel.rayleigh.Rayleigh*

DMRT electromagnetic model in the short range limit (grains AND aggregates are small) as implemented in DMRTML

Parameters

- **sensor** – sensor instance
- **layer** – layer instance

Dense_snow_correction set how snow denser than half the ice density (ie. fractional volume larger than 0.5 is handled).

"auto" means that snow is modeled as air bubble in ice instead of ice spheres in air. "bridging" should be developed in the future.

basic_check()

7.5 smrt.emmodel.dmrt_qcacp_shortrange module

Compute scattering with DMRT QCACP Short range like in DMRT-ML. Short range means that it is accurate only for small and weakly sticky spheres (high stickiness value). It diverges (increasing scattering coefficient) if these conditions are not met. Numerically the size conditions can be evaluated with the ratio radius/wavelength as for Rayleigh scatterers. For the stickiness, it is more difficult as this depends on the size of the scatterers and the fractional volume. In any case, it is dangerous to use too small a stickiness value, especially if the grains are big.

This model is only compatible with the SHS microstructure model.

Examples:

from smrt import make_snowpack, make_sensor

density = [345.0] temperature = [260.0] thickness = [70] radius = [750e-6] stickiness = [0.1]

snowpack = make_snowpack(thickness, "sticky_hard_spheres", density=density, temperature=temperature, radius=radius, stickiness=stickiness)

create the EM Model - Equivalent DMRTML m = make_model("dmrt_shortrange", "dort")

create the sensor theta = np.arange(5.0, 80.0, 5.0) radiometer = sensor.amsre()

class DMRT_QCACP_ShortRange(*sensor*, *layer*, *dense_snow_correction='auto'*)
Bases: *smrt.emmodel.rayleigh.Rayleigh*

DMRT electromagnetic model in the short range limit (grains AND aggregates are small) as implemented in DMRTML

Parameters

- **sensor** – sensor instance
- **layer** – layer instance

Dense_snow_correction set how snow denser than half the ice density (ie. fractional volume larger than 0.5 is handled).

"auto" means that snow is modeled as air bubble in ice instead of ice spheres in air. "bridging" should be developed in the future.

basic_check()

7.6 smrt.emmodel.iba module

Compute scattering from Improved Born Approximation theory as described in Mätzler 1998 and Mätzler and Wiesman 1999, except the absorption coefficient which is computed with Polden von Staten formulation instead of the Eq 24 in Mätzler 1998. See iba_original.py for a fully conforming IBA version.

This model allows for different microstructural models provided that the Fourier transform of the correlation function

may be performed. All properties relate to a single layer.

derived_IBA(*effective_permittivity_model=<function polder_van_santen>*)
return a new IBA model with variant from the default IBA.

Parameters **effective_permittivity_model** – permittivity mixing formula.

:returns a new class inheriting from IBA but with patched methods

class IBA(*sensor*, *layer*)

Bases: *smrt.emmodel.common.AdjustableEffectivePermittivityMixins*

Improved Born Approximation electromagnetic model class.

As with all electromagnetic modules, this class is used to create an electromagnetic object that holds information about the effective permittivity, extinction coefficient and phase function for a particular snow layer. Due to the frequency dependence, information about the sensor is required. Passive and active sensors also have different requirements on the size of the phase matrix as redundant information is not calculated for the passive case.

Parameters

- **sensor** – object containing sensor characteristics
- **layer** – object containing snow layer characteristics (single layer)

Usage Example:

This class is not normally accessed directly by the user, but forms part of the smrt model, together with the radiative solver (in this example, *dort*) i.e.:

```
from smrt import make_model
model = make_model("iba", "dort")
```

iba does not need to be imported by the user due to autoimport of electromagnetic model modules

static effective_permittivity_model(*frac_volume*, *e0=1*, *eps=3.185*, *depol_xyz=None*, *length_ratio=None*, *inclusion_shape=None*, *mixing_ratio=1*)

Calculates effective permittivity of snow by solution of quadratic Polder Van Santen equation for spherical inclusion.

Parameters

- **frac_volume** – Fractional volume of inclusions
- **e0** – Permittivity of background (default is 1)
- **eps** – Permittivity of scattering material (default is 3.185 to compare with MEMLS)
- **depol_xyz** – [Optional] Depolarization factors, spherical isotropy is default. It is not taken into account here.
- **length_ratio** – Length_ratio. Used to estimate depolarization factors when they are not given.
- **inclusion_shape** – Assumption for shape(s) of brine inclusions. Can be a string for single shape, or a list/tuple/dict of strings for mixture of shapes. So far, we have the following shapes: “spheres” and “random_needles” (i.e. randomly-oriented elongated ellipsoidal inclusions). If the argument is a dict, the keys are the shapes and the values are the mixing ratio. If it is a list, the mixing_ratio argument is required.
- **mixing_ratio** – The mixing ratio of the shapes. This is only relevant when inclusion_shape is a list/tuple. Mixing ratio must be a sequence with length len(inclusion_shape)-1. The mixing ratio of the last shapes is deduced as the sum of the ratios must equal to 1.

Returns Effective permittivity

Usage example:

```
from smrt.permittivity.generic_mixing_formula import polder_van_santen
effective_permittivity = polder_van_santen(frac_volume, e0, eps)

# for a mixture of 30% spheres and 70% needles
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_
↪shape={"spheres": 0.3, "random_needles": 0.7})
# or
effective_permittivity = polder_van_santen(frac_volume, e0, eps, inclusion_
↪shape=("spheres", "random_needles"), mixing_ratio=0.3)
```

Todo: Extend Polder Van Santen model to account for ellipsoidal inclusions

compute_iba_coeff()
Calculate angular independent IBA coefficient: used in both scattering coefficient and phase function calculations

Note: Requires mean squared field ratio (uses mean_sq_field_ratio method)

mean_sq_field_ratio(*e0*, *eps*)
Mean squared field ratio calculation

Uses layer effective permittivity

Parameters

- **e0** – background relative permittivity
- **eps** – scattering constituent relative permittivity

basic_check()

compute_ks()
Calculate scattering coefficient: integrate p11+p12 over mu

ks_integrand(*mu*)
This is the scattering function for the IBA model.

It uses the phase matrix in the 1-2 frame. With incident angle chosen to be 0, the scattering angle becomes the scattering zenith angle:

$$\Theta = \theta$$

Scattering coefficient is determined by integration over the scattering angle (0 to pi)

Parameters **mu** – cosine of the scattering angle (single angle)

$$ks\_int = p11 + p22$$

The integration is performed outside this method.

phase(*mu_s*, *mu_i*, *dphi*, *npol=2*)
IBA Phase function (not decomposed).

ft_even_phase(*mu_s*, *mu_i*, *m_max*, *npol=None*)
Calculation of the Fourier decomposed IBA phase function.

This method calculates the Improved Born Approximation phase matrix for all Fourier decomposition modes and return the output.

Coefficients within the phase function are

Passive case (m = 0 only) and active (m = 0)

```
M  = [Pvvp  Pvhp]
     [Phvp  Phhp]
```

Active case (m > 0):

```
M =  [Pvvp Pvhp Pvup]
     [Phvp Phhp Phup]
     [Puvp Puhp Puup]
```

The IBA phase function is given in Mätzler, C. (1998). Improved Born approximation for scattering of radiation in a granular medium. *Journal of Applied Physics*, 83(11), 6111-6117. Here, calculation of the phase matrix is based on the phase matrix in the 1-2 frame, which is then rotated according to the incident and scattering angles, as described in e.g. *Thermal Microwave Radiation: Applications for Remote Sensing, Mätzler (2006)*. Fourier decomposition is then performed to separate the azimuthal dependency from the incidence angle dependency.

Parameters

- **mu_s** – 1-D array of cosine of viewing radiation stream angles (set by solver)
- **mu_i** – 1-D array of cosine of incident radiation stream angles (set by solver)
- **m_max** – maximum Fourier decomposition mode needed
- **npol** – number of polarizations considered (set from sensor characteristics)

compute_ka()

IBA absorption coefficient calculated from the low-loss assumption of a general lossy medium.

Calculates ka from wavenumber in free space (determined from sensor), and effective permittivity of the medium (snow layer property)

Return ka absorption coefficient [m $^{-1}$]

Note: This may not be suitable for high density material

---

ke(*mu*, *npol=2*)

IBA extinction coefficient matrix

The extinction coefficient is defined as the sum of scattering and absorption coefficients. However, the radiative transfer solver requires this in matrix form, so this method is called by the solver.

param mu 1-D array of cosines of radiation stream incidence angles

param npol number of polarization

returns ke extinction coefficient matrix [m $^{-1}$]

Note: Spherical isotropy assumed (all elements in matrix are identical).

Size of extinction coefficient matrix depends on number of radiation streams, which is set by the radiative transfer solver.

---

class IBA_MM(*sensor*, *layer*)

Bases: *smrt.emmodel.iba.IBA*

7.7 smrt.emmodel.iba_original module

Compute scattering from Improved Born Approximation theory. This model allows for different microstructural models provided that the Fourier transform of the correlation function may be performed. All properties relate to a single layer. The absorption is calculated with the original formula in Mätzler 1998

class IBA_original(*sensor*, *layer*)

Bases: *smrt.emmodel.iba.IBA*

Original Improved Born Approximation electromagnetic model class.

As with all electromagnetic modules, this class is used to create an electromagnetic object that holds information about the effective permittivity, extinction coefficient and phase function for a particular snow layer. Due to the frequency dependence, information about the sensor is required. Passive and active sensors also have different requirements on the size of the phase matrix as redundant information is not calculated for the passive case.

Parameters

- **sensor** – object containing sensor characteristics
- **layer** – object containing snow layer characteristics (single layer)

compute_ka()

IBA absorption coefficient calculated from the low-loss assumption of a general lossy medium.

Calculates ka from wavenumber in free space (determined from sensor), and effective permittivity of the medium (snow layer property)

Return ka absorption coefficient [m $^{-1}$]

Note: This may not be suitable for high density material

7.8 smrt.emmodel.nonscattering module

Non-scattering medium can be applied to medium without heteoreneity (like water or pure ice layer).

class NonScattering(*sensor*, *layer*)

Bases: object

basic_check()

ft_even_phase(*mu_s*, *mu_i*, *m_max*, *npol=None*)

Non-scattering phase matrix.

Returns : null phase matrix

phase(*mu_s*, *mu_i*, *dphi*, *npol=2*)

Non-scattering phase matrix.

Returns : null phase matrix

ke(*mu*, *npol=2*)

effective_permittivity()

7.9 smrt.emmodel.prescribed_kskaeps module

Use prescribed scattering ks and absorption ka coefficients and effective permittivity in the layer. The phase matrix has the Rayleigh form with prescribed scattering coefficient

This model is compatible with any microstructure but requires that ks, ka, and optionally effective permittivity to be set in the layer

Example:

```
m = make_model("prescribed_kskaeps", "dort")
snowpack.layers[0].ks = ks
snowpack.layers[0].ka = ka
snowpack.layers[0].effective_permittivity = eff_eps
```

class Prescribed_KsKaEps(*sensor*, *layer*)

Bases: *smrt.emmodel.rayleigh.Rayleigh*

7.10 smrt.emmodel.rayleigh module

Compute Rayleigh scattering. This theory requires the scatterers to be smaller than the wavelength and the medium to be sparsely populated (eq. very low density in the case of snow).

This model is only compatible with the Independent Sphere microstructure model

class Rayleigh(*sensor*, *layer*)

Bases: object

basic_check()

ft_even_phase_baseonUlaby(*mu_s*, *mu_i*, *m_max*, *npol=None*)

Equations are from pg 1188-1189 Ulaby, Moore, Fung. Microwave Remote Sensing Vol III. # See also pg 157 of Tsang, Kong and Shin: Theory of Microwave Remote Sensing (1985) - can be used to derive # the Ulaby equations.

ft_even_phase_basedonJin(*mu_s*, *mu_i*, *m_max*, *npol=None*)

Rayleigh phase matrix.

These are the Fourier decomposed phase matrices for modes m = 0, 1, 2. It is based on Y.Q. Jin

Coefficients within the phase function are:

M = [Pvvp Pvhp] [Phvp Phhp]

Inputs are: :param m: mode for decomposed phase matrix (0, 1, 2) :param mu: vector of cosines of incidence angle

Returns P: phase matrix

ft_even_phase_tsang(*mu_s*, *mu_i*, *m_max*, *npol=None*)

Rayleigh phase matrix.

These are the Fourier decomposed phase matrices for modes m = 0, 1, 2. Equations are from p128 Tsang Application and Theory 200 and sympy calculations

Coefficients within the phase function are:

M = [P[v, v] P[v, h] -P[v, u]] [P[h, v] P[h, h] -P[h, u]] [P[u, v] P[u, h] P[u, u]]

Inputs are: :param m: mode for decomposed phase matrix (0, 1, 2) :param mu: vector of cosines of incidence angle

Returns P: phase matrix

ft_even_phase(*mu_s*, *mu_i*, *m_max*, *npol=None*)
Equations are from pg 1188-1189 Ulaby, Moore, Fung. Microwave Remote Sensing Vol III. # See also pg 157 of Tsang, Kong and Shin: Theory of Microwave Remote Sensing (1985) - can be used to derive # the Ulaby equations.

phase(*mu_s*, *mu_i*, *dphi*, *npol=2*)

ke(*mu*, *npol=2*)
return the extinction coefficient matrix

The extinction coefficient is defined as the sum of scattering and absorption coefficients. However, the radiative transfer solver requires this in matrix form, so this method is called by the solver.

param mu 1-D array of cosines of radiation stream incidence angles

param npol number of polarizations

returns ke extinction coefficient matrix [m $^{-1}$]

Note: Spherical isotropy assumed (all elements in matrix are identical).

Size of extinction coefficient matrix depends on number of radiation streams, which is set by the radiative transfer solver.

---

effective_permittivity()

7.11 smrt.emmodel.sft_rayleigh module

Compute Strong Fluctuation Theory scattering. This theory requires the scatterers to be smaller than the wavelength

This model is only compatible with the Exponential autocorrelation function only

class SFT_Rayleigh(*sensor*, *layer*)
Bases: *smrt.emmodel.rayleigh.Rayleigh*

7.12 smrt.emmodel.test_iba module

setup_func_sp()

setup_func_indep(*radius=0.0005*)

setup_func_shs()

setup_func_pc(*pc*)

setup_func_em(*testpack=None*)

setup_func_active(*testpack=None*)

setup_func_rayleigh()

setup_mu(*stepsize*, *bypass_exception=None*)

test_ks_pc_is_0p3_mm()

test_ks_pc_is_0p25_mm()

test_ks_pc_is_0p15_mm()

test_ks_pc_is_0p1_mm()

test_ks_pc_is_0p2_mm()

test_energy_conservation_exp()

test_energy_conservation_indep()

test_energy_conservation_shs()

test_npol_passive_is_2()

test_npol_active_is_3()

test_energy_conservation_exp_active()

test_energy_conservation_indep_active()

test_energy_conservation_shs_active()

test_iba_vs_rayleigh_passive_m0()

test_iba_vs_rayleigh_active_m0()

test_iba_vs_rayleigh_active_m1()

test_iba_vs_rayleigh_active_m2()

test_permittivity_model()

test_iba_raise_exception_mu_is_1()

7.13 smrt.emmodel.test_iba_original module

setup_func_sp()

setup_func_indep(*radius=0.0005*)

setup_func_shs()

setup_func_pc(*pc*)

setup_func_em(*testpack=None*)

setup_func_active(*testpack=None*)

setup_func_rayleigh()

setup_mu(*stepsize*, *bypass_exception=None*)

test_ks_pc_is_0p3_mm()

test_ks_pc_is_0p25_mm()

test_ks_pc_is_0p15_mm()

test_ks_pc_is_0p1_mm()

test_ks_pc_is_0p2_mm()

test_energy_conservation_exp()

test_energy_conservation_indep()

test_energy_conservation_shs()

test_npol_passive_is_2()

test_npol_active_is_3()

test_energy_conservation_exp_active()

test_energy_conservation_indep_active()

test_energy_conservation_shs_active()

test_iba_vs_rayleigh_passive_m0()

test_iba_vs_rayleigh_active_m0()

test_iba_vs_rayleigh_active_m1()

test_iba_vs_rayleigh_active_m2()

test_iba_raise_exception_mu_is_1()

7.14 smrt.emmodel.test_prescribed_kskaeps module

setup_func_sp()

setup_func_em(*testpack=None*)

test_energy_conservation()

7.15 smrt.emmodel.test_rayleigh module

setup_func_sp()

setup_func_em(*testpack=None*)

test_energy_conservation()

test_energy_conservation_tsang()

test_energy_conservation_jin()

7.16 smrt.emmodel.test_sft_rayleigh module

setup_func_sp()

setup_func_em(*testpack=None*)

test_energy_conservation()

7.17 Module contents

This directory contains the different electromagnetic (EM) models that compute the scattering and absorption coefficients and the phase function in a _given_ _layer_. The computation of the inter-layer propagation is done by the *rtsolver* package.

The EM models differ in many aspects, one of which is the constraint on the microstructure model they can be used with. The *smrt.emmodel.iba* model can use any microstructure model that defines autocorrelation functions (or its FT). In contrast others such as `smrt.emmodel.dmrt_shortrange` is bound to the `smrt.microstructuremodel.sticky_hard_spheres` microstructure for theoretical reasons.

The selection of the EM model is done with the *smrt.core.model.make_model()* function

---

For developers

To implement a new scattering formulation / phase function, we recommend to start from an existing module, probably rayleigh.py is the simplest. Copy this file to *myscatteringtheory.py* or any meaningful name. It can be directly used with *make_model()* function as follows:

```
m = make_model("myscatteringtheory", "dort")
```

Note that if the file is not in the emmodels directory, you must explicitly import the module and pass it to make_model as a module object (instead of a string).

An emmodel model must define:

- ks and ka attributes/properties
- ke() and effective_permittivity() methods
- at least one of the phase and ft_even_phase methods (both is better).

For the details it is recommended to contact the authors as the calling arguments and required methods may change time to time.

CHAPTER 8

smrt.rtsolver package

8.1 Submodules

8.2 smrt.rtsolver.dort module

The Discrete Ordinate and Eigenvalue Solver is a multi-stream solver of the radiative transfer model. It is precise but less efficient than 2 or 6 flux solvers. Different flavours of DORT (or DISORT) exist depending on the mode (passive or active), on the density of the medium (sparse media have trivial inter-layer boundary conditions), on the way the streams are connected between the layers and on the way the phase function is prescribed. The actual version is a blend between Picard et al. 2004 (active mode for sparse media) and DMRT-ML (Picard et al. 2013) which works in passive mode only for snow. The DISORT often used in optics (Stamnes et al. 1988) works only for sparse medium and uses a development of the phase function in Legendre polynomia on theta. The version used in DMRT-QMS (L. Tsang's group) is similar to the present implementation except it uses spline interpolation to connect constant-angle streams between the layers although we use direct connection by varying the angle according to Snell's law. A practical consequence is that the number of streams vary (due to internal reflection) and the value *n_max_stream* only applies in the most refringent layer. The number of outgoing streams in the air is usually smaller, sometimes twice smaller (depends on the density profile). It is important not to set too low a value for n_max_streams. E.g. 32 is usually fine, 64 or 128 are better but simulations will be much slower.

class DORT(*n_max_stream=32*, *m_max=2*, *stream_mode='most_refringent'*, *phase_normalization=True*, *error_handling='exception'*, *process_coherent_layers=False*, *prune_deep_snowpack=None*)

Bases: `object`

Discrete Ordinate and Eigenvalue Solver

Parameters

- **n_max_stream** – number of stream in the most refringent layer
- **m_max** – number of mode (azimuth)
- **phase_normalization** – the integral of the phase matrix should in principe be equal to the scattering coefficient.

However, some emmodels do not respect this strictly. In general a small difference is due to numerical rounding and is acceptable, but a large difference rather indicates either a bug in the emmodel or input parameters that breaks the assumption of the emmodel. The most typical case is when the grain size is too big compared to wavelength for emmodels that rely on Rayleigh assumption. If this argument is to True (the default), the phase matrix is normalized to be coherent with the scattering coefficient, but only when the difference is moderate (0.7 to 1.3). If set to "force" the normalization is always performed. This option is dangerous because it may hide bugs or unappropriate input parameters (typically too big grains). If set to False, no normalization is performed. :param error_handling: If set to "exception" (the default), raise an exception in cause of error, stopping the code. If set to "nan", return a nan, so the calculation can continue, but the result is of course unusuable and the error message is not accessible. This is only recommended for long simulations that sometimes produce an error. :param process_coherent_layers: Adapt the layers thiner than the wavelegnth using the MEMLS method. The radiative transfer theory is inadequate layers thiner than the wavelength and using DORT with thin layers is generally not recommended. In some parcticular cases (such as ice lenses) where the thin layer is isolated between large layers, it is possible to replace the thin layer by an equivalent reflective interface. This neglects scattering in the thin layer, which is acceptable in most case, because the layer is thin. To use this option and more generally to investigate ice lenses, it is recommended to read MEMLS documentation on this topic. :param prune_deep_snowpack: this value is the optical depth from which the layers are discarded in the calculation. It is to be use to accelerate the calculations for deep snowpacks or at high frequencies when the contribution of the lowest layers is neglegible. The optical depth is a good criteria to determine this limit. A value of about 6 is recommended. Use with care, especially values lower than 6.

solve(*snowpack*, *emmodels*, *sensor*, *atmosphere=None*)
solve the radiative transfer equation for a given snowpack, emmodels and sensor configuration.

dort(*m_max=0*, *special_return=False*)

prepare_intensity_array(*streams*)

dort_modem_banded(*m*, *streams*, *eigenvalue_solver*, *interfaces*, *intensity_down_m*, *compute_coherent_only=False*, *special_return=False*)

muleye(*x*)

matmul(*a*, *b*, **args*)

compiled_todiag

todiag(*bmat*, *oi*, *oj*, *dmat*)

extend_2pol_npol(*x*, *npol*)

class EigenValueSolver(*ke*, *ks*, *ft_even_phase_function*, *mu*, *weight*, *m_max*, *normalization*)
Bases: `object`

solve(*m*, *compute_coherent_only*, *debug_A=False*)

normalize(*m*, *A*)

class InterfaceProperties(*frequency*, *interfaces*, *substrate*, *permittivity*, *streams*, *m_max*, *npol*)
Bases: `object`

reflection_top(*l*, *m*, *compute_coherent_only*)

reflection_bottom(*l*, *m*, *compute_coherent_only*)

transmission_top(*l*, *m*, *compute_coherent_only*)

transmission_bottom(*l*, *m*, *compute_coherent_only*)

static combine_coherent_diffuse_matrix(*coh*, *diff*, *m*, *compute_coherent_only*)

normalize_diffuse_matrix(*mat*, *mu_st*, *mu_i*, *weights*)

class Streams
Bases: object

compute_stream(*n_max_stream*, *permittivity*, *permittivity_substrate*, *mode='most_refringent'*)

gaussquad(*n*)

8.3 smrt.rtsolver.dort_nonormalization module

The Discrete Ordinate and Eigenvalue Solver is a multi-stream solver of the radiative transfer model. It is precise but less efficient than 2 or 6 flux solvers. Different flavours of DORT (or DISORT) exist depending on the mode (passive or active), on the density of the medium (sparse media have trivial inter-layer boundary conditions), on the way the streams are connected between the layers and on the way the phase function is prescribed. The actual version is a blend between Picard et al. 2004 (active mode for sparse media) and DMRT-ML (Picard et al. 2013) which works in passive mode only for snow. The DISORT often used in optics (Stamnes et al. 1988) works only for sparse medium and uses a development of the phase function in Legendre polynomia on theta. The version used in DMRT-QMS (L. Tsang's group) is similar to the present implementation except it uses spline interpolation to connect constant-angle streams between the layers although we use direct connection by varying the angle according to Snell's law. A practical consequence is that the number of streams vary (due to internal reflection) and the value *n_max_stream* only applies in the most refringent layer. The number of outgoing streams in the air is usually smaller, sometimes twice smaller (depends on the density profile). It is important not to set too low a value for n_max_stream. E.g. 32 is usually fine, 64 or 128 are better but simulations will be much slower.

class DORT(*n_max_stream=32*, *m_max=2*, *stream_mode='most_refringent'*)
Bases: object

Discrete Ordinate and Eigenvalue Solver

Parameters

- **n_max_stream** – number of stream in the most refringent layer
- **m_max** – number of mode (azimuth)

solve(*snowpack*, *emmodels*, *sensor*, *atmosphere=None*)
solve the radiative transfer equation for a given snowpack, emmodels and sensor configuration.

dort(*m_max=0*, *special_return=False*)

prepare_intensity_array(*outmu*, *outweight*)

dort_modem_banded(*m*, *n_stream*, *mu*, *weight*, *outmu*, *n_stream_substrate*, *intensity_down_m*, *compute_coherent_only=False*, *special_return=False*)

fix_matrix(*x*)

muleye(*x*)

todiag(*bmat*, *ij*, *dmat*)

extend_2pol_npol(*x*, *npol*)

solve_eigenvalue_problem(*m*, *ke*, *ft_even_phase*, *mu*, *weight*)

compute_stream(*n_max_stream*, *permittivity*, *permittivity_substrate*, *mode='most_refringent'*)

gaussquad(*n*)

8.4 smrt.rtsolver.nadir_lrm_altimetry module

class NadirLRMAltimetry(*waveform_model=None*, *oversampling=10*, *return_oversampled=False*, *skip_pfs_convolution=False*, *return_contributions=False*, *theta_inc_sampling=1*, *return_theta_inc_sampling=False*, *error_handling='exception'*)

Bases: `object`

Solver based on Adams and Brown 1998 and Lacroix et al. 2008. Both models differ in the specific choices for the scattering and backscatter of the interface, but are similar in the way the waveform is calculated, which concerns the solver here.

Parameters **oversampling** – integer number defining the number of subgates used for the computation in each altimeter gate.

This is equivalent to multiply the bandwidth by this number. It is used to perform more accurate computation. :param return_oversampled: by default the backscatter is returned for each gate. If set to True, the oversampled waveform is returned instead. See the 'oversampling' argument. :param return_contributions: return volume, surface and interface backscatter contributions in addition to the total backscatter.

solve(*snowpack*, *emmodels*, *sensor*, *atmosphere=None*)
solve the radiative transfer equation for a given snowpack, emmodels and sensor configuration.

convolve_with_PFS_PTR_PDF(*t_gate*, *backscatter*, *t_inc_sample*)

gate_depth(*eps=None*)
return gate depth that cover the snowpack for a regular time sampling

combined_depth_grid()

vertical_scattering_distribution(*return_contributions*, *mu_i=1.0*)

Compute the vertical backscattering distribution due to "grain" or volume scattering (symbol pvg in Eq 9 in Lacroix 2008) and

"interfaces" or 'surface' scattering (symbol pvl in Eq 9 in Lacroix 2008)

param mu cosine of the incidence angles. Only the dependence on the surface scattering depend on mu_i

PFS_numerical(*tau*)

fill_forward(*a*, *where*, *axis=-1*)

fill(*a*, *where*, *novalue=0*)

8.5 smrt.rtsolver.test_dort module

setup_snowpack()

setup_snowpack_with_DH()

setup_2layer_snowpack()

test_noabsoprtion()

test_returned_theta()

test_selectby_theta()

test_depth_hoar_stream_numbers()

test_2layer_pack()

test_shallow_snowpack()

8.6 smrt.rtsolver.waveform_model module

class WaveformModel
Bases: object

class Brown1977(*sensor*, *numerical_convolution=False*)
Bases: *smrt.rtsolver.waveform_model.WaveformModel*

Antenna Gain formulation used by Brown 1977. The formula is exp(2/gamma * sin(theta)**2) for the perfect nadir case, but is also available with off-nadir angles.

G(*theta*, *phi*)

PFS(*tau*, *surface_slope=0*, *shift_nominal_gate=True*)

PFS_PTR_PDF(*tau*, *sigma_surface=0*, *surface_slope=0*)
compute the convolution of the PFS and PTR

Parameters

- **sensor** – sensor to apply the antenna gain
- **tau** – time to which to compute the PFSxPTR
- **sigma_surface** – RMS height of the surface topography (meter)

class Newkrik1992(*sensor*)
Bases: *smrt.rtsolver.waveform_model.WaveformModel*

Antenna Gain formulation proposed by Newkrik and Brown, 1992. Compared to the classical Bronw 1977, it takes into account the asymmetry of the antenna pattern in the co and cross-track direction.

G(*theta*, *phi*)

PFS(*sensor*, *tau*)

8.7 Module contents

This directory contains different solvers of the radiative transfer equation. Based on the electromagnetic properties of each layer computed by the EM model, these RT solvers compute the emission and propagation of energy in the medium up to the surface (the atmosphere is usually dealt with independently in dedicated modules in *smrt.atmosphere*).

The solvers differ by the approximations and numerical methods. *dort* is currently the most accurate and recommended in most cases unless the computation time is a constraint.

The selection of the solver is done with the *make_model()* function.

For Developers

To experiment with DORT, we recommand to copy the file dort.py to e.g. dort_mytest.py so it is immediately available through *make_model()*.

To develop a new solver that will be accessible by the *make_model()* function, you need to add a file in this directory, give a look at dort.py which is not simple but the only one at the moment. Only the method solve needs to be implemented. It must return a *Result* instance with the results. Contact the core developers to have more details.

CHAPTER 9

smrt.core package

9.1 Submodules

9.2 smrt.core.atmosphere module

class AtmosphereBase
Bases: object

9.3 smrt.core.error module

Definition of the Exception specific to SMRT.

exception SMRTError
Bases: Exception

Error raised by the model

exception SMRTWarning
Bases: Warning

Warning raised by the model

smrt_warn(*message*)

9.4 smrt.core.filelock module

A platform independent file lock that supports the with-statement.

exception Timeout(*lock_file*)
Bases: TimeoutError

Raised when the lock could not be acquired in *timeout* seconds.

lock_file = None
The path of the file lock.

class BaseFileLock(*lock_file*, *timeout=-1*)
Bases: object

Implements the base class of a file lock.

lock_file
The path to the lock file.

timeout
You can set a default timeout for the filelock. It will be used as fallback value in the acquire method, if no timeout value (*None*) is given.

If you want to disable the timeout, set it to a negative value.

A timeout of 0 means, that there is exactly one attempt to acquire the file lock.

New in version 2.0.0.

is_locked
True, if the object holds the file lock.

Changed in version 2.0.0: This was previously a method and is now a property.

acquire(*timeout=None*, *poll_intervall=0.05*)
Acquires the file lock or fails with a *Timeout* error.

```
# You can use this method in the context manager (recommended)
with lock.acquire():
    pass

# Or use an equivalent try-finally construct:
lock.acquire()
try:
    pass
finally:
    lock.release()
```

Parameters

- **timeout** (*float*) – The maximum time waited for the file lock. If timeout < 0, there is no timeout and this method will block until the lock could be acquired. If timeout is None, the default *timeout* is used.
- **poll_intervall** (*float*) – We check once in *poll_intervall* seconds if we can acquire the file lock.

Raises ***Timeout*** – if the lock could not be acquired in *timeout* seconds.

Changed in version 2.0.0: This method returns now a *proxy* object instead of *self*, so that it can be used in a with statement without side effects.

release(*force=False*)
Releases the file lock.

Please note, that the lock is only completly released, if the lock counter is 0.

Also note, that the lock file itself is not automatically deleted.

Parameters **force** (*bool*) – If true, the lock counter is ignored and the lock is released in every case.

class WindowsFileLock(*lock_file*, *timeout=-1*)
Bases: *smrt.core.filelock.BaseFileLock*

Uses the `msvcrt.locking()` function to hard lock the lock file on windows systems.

class UnixFileLock(*lock_file*, *timeout=-1*)
Bases: *smrt.core.filelock.BaseFileLock*

Uses the `fcntl.flock()` to hard lock the lock file on unix systems.

class SoftFileLock(*lock_file*, *timeout=-1*)
Bases: *smrt.core.filelock.BaseFileLock*

Simply watches the existence of the lock file.

FileLock
Alias for the lock, which should be used for the current platform. On Windows, this is an alias for *WindowsFileLock*, on Unix for *UnixFileLock* and otherwise for *SoftFileLock*.

alias of *smrt.core.filelock.UnixFileLock*

9.5 smrt.core.fresnel module

fresnel coefficients formulae used in the packages *smrt.interface* and *smrt.substrate*.

fresnel_coefficients_old(*eps_1*, *eps_2*, *mu1*)
compute the reflection in two polarizations (H and V). The equations are only valid for lossless media. Applying these equations for (strongly) lossy media result in (large) errors. Don't use it. It is here for reference only.

Parameters

- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns rv, rh, mu2 the cosine of the angle in medium 2

fresnel_coefficients_maezawa09_classical(*eps_1*, *eps_2*, *mu1*, *full_output=False*)
compute the reflection in two polarizations (H and V) for lossly media with the "classical Fresnel" based on Maezawa, H., & Miyauchi, H. (2009). Rigorous expressions for the Fresnel equations at interfaces between absorbing media. Journal of the Optical Society of America A, 26(2), 330. https://doi.org/10.1364/josaa.26.000330

The classical derivation does not respect energy conservation, especially the transmittivity. Don't use it. It is here for reference only.

Parameters

- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns rv, rh, mu2 the cosine of the angle in medium 2

fresnel_coefficients_maezawa09_rigorous(*eps_1*, *eps_2*, *mu1*, *full_output=False*)
compute the reflection in two polarizations (H and V) for lossly media with the "rigorous Fresnel" based on Maezawa, H., & Miyauchi, H. (2009). Rigorous expressions for the Fresnel equations at interfaces between absorbing media. Journal of the Optical Society of America A, 26(2), 330. https://doi.org/10.1364/josaa.26.000330

The 'rigorous' derivation respect the energy conservation even for strongly loosly media.

Parameters

- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns rv, rh, mu2 the cosine of the angle in medium 2

fresnel_coefficients(*eps_1*, *eps_2*, *mu1*, *full_output=False*)

compute the reflection in two polarizations (H and V) for lossly media with the "rigorous Fresnel" based on Maezawa, H., & Miyauchi, H. (2009). Rigorous expressions for the Fresnel equations at interfaces between absorbing media. Journal of the Optical Society of America A, 26(2), 330. https://doi.org/10.1364/josaa.26.000330

The 'rigorous' derivation respect the energy conservation even for strongly loosly media.

Parameters

- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns rv, rh, mu2 the cosine of the angle in medium 2

brewster_angle(*eps_1*, *eps_2*)

compute the brewster angle

Parameters

- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.

Returns angle in radians

fresnel_reflection_matrix(*eps_1*, *eps_2*, *mu1*, *npol*)

compute the fresnel reflection matrix for/in medium 1 laying above medium 2.

Parameters

- **npol** – number of polarizations to return.
- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns a matrix or the diagional depending on *return_as_diagonal*

fresnel_transmission_matrix(*eps_1*, *eps_2*, *mu1*, *npol*)

compute the fresnel reflection matrix for/in medium 1 laying above medium 2.

Parameters

- **npol** – number of polarizations to return.
- **eps_1** – permittivity of medium 1.
- **eps_2** – permittivity of medium 2.
- **mu1** – cosine zenith angle in medium 1.

Returns a matrix or the diagional depending on *return_as_diagonal*

9.6 smrt.core.globalconstants module

Global constants used throughout the model are defined here and imported as needed. The constants are:

Parameter	Description	Value
DENSITY_OF_ICE	Density of pure ice at 273.15K	916.7 kg m $^{-3}$
FREEZING_POINT	Freezing point of pure water	273.15 K
C_SPEED	Speed of light in a vacuum	2.99792458 x 10 ^{8} ms $^{-1}$
PERMITTIVITY_OF_AIR	Relative permittivity of air	1

Usage example:

```
from smrt.core.globalconstants import DENSITY_OF_ICE
```

9.7 smrt.core.interface module

This module implements the base class for all the substrate models. To create a substrate, it is recommended to use help functions such as `make_soil()` rather than the class constructor.

make_interface(*inst_class_or_modulename*, *broadcast=True*, ***kwargs*)
return an instance corresponding to the interface model with the provided arguments.

This function imports the interface module if necessary and return an instance of the interface class with the provided arguments in **kwargs.

Parameters

- **inst_class_or_modulename** – a class, and instance or the name of the python module in smrt/interface
- ****kwargs** – all the arguments required by the interface class

class Interface(***kwargs*)
Bases: `object`

Abstract class for interface between layer and substrate at the bottom of the snowpack. It provides argument handling.

args = []

optional_args = {}

class SubstrateBase(*temperature=None*, *permittivity_model=None*)
Bases: `object`

Abstract class for substrate at the bottom of the snowpack. It provides calculation of the permittivity constant for soil case. Argument handline is delegated to the instance of the interface

permittivity(*frequency*)
compute the permittivity for the given frequency using permittivity_model. This method returns None when no permittivity model is available. This must be handled by the calling code and interpreted suitably.

substrate_from_interface(*interface_cls*)
this decorator transform an interface class into a substrate class with automatic method

class Substrate(*temperature=None*, *permittivity_model=None*, ***kwargs*)
Bases: *smrt.core.interface.SubstrateBase*, *smrt.core.interface.Interface*

get_substrate_model(*substrate_model*)
return the class corresponding to the substrate model called name. This function imports the correct module if possible and returns the class

9.8 smrt.core.layer module

Layer instance contains all the properties for a single snow layer (e.g. temperature, frac_volume, etc). It also contains a *microstructure* attribute that holds the microstructural properties (e.g. radius, corr_length, etc). The class of this attribute defines the microstructure model to use (see *smrt.microstructure_model* package).

To create a single layer, it is recommended to use the function *make_snow_layer()* rather than the class constructor. However it is usually more convenient to create a snowpack using *make_snowpack()*.

For developers

The *Layer* class should not be modified at all even if you need new properties to define the layer (e.g. brine concentration, humidity, ...). If the property you need to add is related to geometric aspects, it is probably better to use an existing microstructure model or to create a new one. If the new parameter is not related to geometrical aspect, write a function similar to *make_snow_layer()* (choose an explicit name for your purpose). In this function, create the layer by calling the Layer constructor as in *make_snow_layer()* and then add your properties with lay.myproperty=xxx, ... See the example of liquid water in *make_snow_layer()*. This approach avoids specialization of the Layer class. The new function can be in any file (inc. out of smrt directories), and should be added in *make_medium* if it is of general interest and written in a generic way, that is, covers many use cases for many users with default arguments, etc.

---

class Layer(*thickness*, *microstructure_model=None*, *temperature=273.15*, *permittivity_model=None*, *inclusion_shape=None*, ***kwargs*)
Bases: `object`

Contains the properties for a single layer including the microstructure attribute which holds the microstructure properties.

To create layer, it is recommended to use of the functions `make_snow_layer()` and similar

ssa
return the SSA, compute it if necessary

frac_volume

permittivity(*i*, *frequency*)
return the permittivity of the i-th medium depending on the frequency and internal layer properties. Usually i=0 is air and i=1 is ice for dry snow with a low or moderate density.

Parameters

- **i** – number of the medium. 0 is reserved for the background
- **frequency** – frequency of the wave (Hz)

Returns complex permittivity of the i-th medium

basic_checks()
Function to provide very basic input checks on the layer information

Currently checks:

- temperature is between 100 and the freezing point (Kelvin units check),
- density is between 1 and DENSITY_OF_ICE (SI units check)
- layer thickness is above zero

inverted_medium()
return the layer with inverted autocorrelation and inverted permittivities.

update(***kwargs*)
update the attributes. This method is to be used when recalculation of the state of the object is necessary. See for instance *SnowLayer*.

get_microstructure_model(*modulename*, *classname=None*)
return the class corresponding to the microstructure_model defined in modulename.

This function import the correct module if possible and return the class. It is used internally and should not be needed for normal usage.

Parameters **modulename** – name of the python module in smrt/microstructure_model

make_microstructure_model(*modelname_or_class*, ***kwargs*)
create an microstructure instance.

This function is called internally and should not be needed for normal use.

param modelname_or_class name of the module or directly the class.

param type string

param **kwargs all the arguments need for the specific autocorrelation.

returns instance of the autocorrelation *modelname* with the parameters given in ***kwargs*

Example

To import the StickyHardSpheres class with spheres radius of 1mm, stickiness of 0.5 and fractional_volume of 0.3:

```
shs = make_autocorrelation("StickyHardSpheres", radius=0.001, stickiness=0.5,␣
↪frac_volume=0.3)
```

layer_properties(**required_arguments*, *optional_arguments=None*, ***kwargs*)
This decorator is used for the permittivity functions (or any other functions) to inject layer's attributes as arguments. The decorator declares the layer properties needed to call the function and the optional ones. This allows permittivity functions to use any property of the layer, as long as it is defined.

9.9 smrt.core.lib module

get(*x*, *i*, *name=None*)

check_argument_size(*x*, *n*, *name=None*)

is_sequence(*x*)

len_atleast_1d(*x*)

class smrt_diag(*arr*)
Bases: object

Scipy.sparse is very slow for diagonal matrix and numpy has no good support for linear algebra. This diag class implements simple diagonal object without numpy subclassing (but without much features). It seems that proper subclassing numpy and overloading matmul is a very difficult problem.

diagonal()

shape

check_type(*other*)

class smrt_matrix(*mat*, *mtype=None*)
Bases: object

SMRT uses two formats of matrix: one most suitable to implement emmodel where equations are different for each polarization and another one suitable for DORT computation where stream and polarization are collapsed in one dimension to allow matrix operation. In addition, the reflection and transmission matrix are often diagonal matrix, which needs to be handled because it saves space and allow much faster operations. This class implemented all these features.

static empty(*dims*, *mtype=None*)

static zeros(*dims*, *mtype=None*)

static ones(*dims*, *mtype=None*)

static full(*dims*, *value*, *mtype=None*)

npol

isnull()

compress(*mode=None*, *auto_reduce_npol=False*)
compress a matrix. This comprises several actions: 1) select one mode, if relevant (dense5, and diagonal5). 2) reduce the number of polarization from 3 to 2 if mode==0 and auto_reduce_npol=True. 3) convert the format of the matrix to compressed numpy, involving a change of the dimension order (pola and streams are merged).

diagonal

sel(***kwargs*)

isnull(*m*)
return true if the smrt matrix is null

generic_ft_even_matrix(*phase_function*, *m_max*, *nsamples=None*)
Calculation of the Fourier decomposed of the phase or reflection or transmission matrix provided by the function.

This method calculates the Fourier decomposition modes and return the output.

Coefficients within the phase function are

Passive case (m = 0 only) and active (m = 0)

```
M  = [Pvvp  Pvhp]
     [Phvp  Phhp]
```

Active case (m > 0):

```
M =  [Pvvp Pvhp Pvup]
     [Phvp Phhp Phup]
     [Puvp Puhp Puup]
```

Parameters

- **phase_function** – must be a function taking dphi as input. It is assumed that phi is symmetrical (it is in cos(phi))
- **m_max** – maximum Fourier decomposition mode needed

set_max_numerical_threads(*nthreads*)
set the maximum number of threads for a few known library. This is useful to disable parallel computing in SMRT when using parallel computing to call multiple // SMRT runs. This avoid over-committing the CPUs and results in much better performance. Inspire from joblib.

cached_roots_legendre(*n*)
Cache roots_legendre results to speed up calls of the fixed_quad function.

9.10 smrt.core.model module

A model in SMRT is composed of the electromagnetic scattering theory (*smrt.emmodel*) and the radiative transfer solver (*smrt.rtsolver*). The *smrt.emmodel* is responsible for computation of the scattering and absorption coefficients and the phase function of a layer. It is applied to each layer and it is even possible to choose different emmodel for each layer (for instance for a complex medium made of different materials: snow, soil, water, atmosphere, ...). The *smrt.rtsolver* is responsible for propagation of the incident or emitted energy through the layers, up to the surface, and eventually through the atmosphere.

To build a model, use the *make_model()* function with the type of emmodel and type of rtsolver as arguments. Then call the *Model.run()* method of the model instance by specifying the sensor (*smrt.core.sensor.Sensor*), snowpack (*smrt.core.snowpack.Snowpack*) and optionally atmosphere (see *smrt.atmosphere*). The results are returned as a *Result* which can then been interrogated to retrieve brightness temperature, backscattering coefficient, etc.

Example:

```
m = make_model("iba", "rtsolver")

result = m.run(sensor, snowpack)  # sensor and snowpack are created before

print(result.TbV())
```

The model can be run on a list of snowpacks or even more conveniently on a *pandas.Series* or *pandas.DataFrame* including snowpacks. The first advantage is that by setting parallel_computation=True, the *Model.run()* method performs the simulation in parallel

on all the available cores of your machine and even possibly remotely on a high performance cluster using dask. The second advantage is that the returned *Result* object contains all the simulations and provide an easier way to plot the results or compute statistics.

If a list of snowpacks is provided, it is recommended to also set the snowpack_dimension argument. It takes the form of a tuple
(list of snowpack_dimension values, dimension name). The name and values are used to define the coordinates in the *Result* object. This is useful with timeseries or sensitivity analysis for instance.

Example:

```
snowpacks = []
times = []
for file in filenames:
    #  create a snowpack for each time series
    sp = ...
    snowpacks.append(sp)
```

(continues on next page)

(continued from previous page)

```
    times.append(sp)

# now run the model

res = m.run(sensor, snowpacks, snowpack_dimension=('time', times))
```

The *res* variable has now a coordinate *time* and res.TbV() returns a timeseries.

Using *pandas.Series* offers an even more elegant way to run SMRT and assemble the results of all the simulations.

> thickness_list = np.arange(0, 10, 1) snowpacks = pd.Series([make_snowpack(thickness=t, ……..) for t in thickness_list], index=thickness_list) # snowpacks is a pandas Series of snowpack objects with the thickness as index
>
> # now run the model
>
> res = m.run(sensor, snowpacks, parallel_computation=True)
>
> # convert the result into a datframe res = res.to_dataframe()

The *res* variable is a dataframe with the thickness as index and the channels of the sensor as column.

Using *pandas.DataFrame* is similar. One column must contain Snowpack objects (see snowpack_column argument). The results of the simulations are automatically joined with this dataframe and returned by `to_dataframe()` or `to_dataframe()`.

> # df is a DataFrame with several parameters in each row.
>
> # add a snowpack object for each row df[‘snowpack’] = [make_snowpack(thickness=row[‘thickness’], ……..) for i, row in df.iterrows()]]
>
> # now run the model res = m.run(sensor, snowpacks, parallel_computation=True)
>
> # convert the result into a datframe res = res.to_dataframe()

The *res* variable is a *pandas.DataFrame* equal to df + the results at all sensor’s channel added.

make_model(*emmodel*, *rtsolver=None*, *emmodel_options=None*, *rtsolver_options=None*, *emmodel_kwargs=None*, *rtsolver_kwargs=None*)

create a new model with a given EM model and RT solver. The model is then ready to be run using the *Model.run()* method. This function is the privileged way to create models compared to class instantiation. It supports automatic import of the emmodel and rtsolver modules.

Parameters **emmodel** – type of emmodel to use. Can be given by the name of a file/module in the emmodel directory (as a string) or a class.

List (and dict, respectively) can be provided when a different emmodel is needed for every layer (or every kind of layer medium). :type emmodel: string or class or list of strings or classes or dict of strings or classes. If a list of emmodels is given, the size must be the same as the number of layers in the snowpack. If a dict is given, the keys are the kinds of medium and the values are the associated emmodels to each sort of medium. The layer attribute ‘medium’ is used to determine the emmodel to use for each layer. :type emmodel: string or class; or list of strings or classes; or dict of strings or classes. :param rtsolver: type of RT solver to use. Can be given by the name of a file/module in the rtsolver directeory (as a string) or a class. :type rtsolver: string or class. Can be None when only computation of the layer electromagnetic properties is needed. :param emmodel_options: extra arguments to use to create emmodel instance. Valid arguments depend on the selected emmodel. It is documented in for each emmodel class. :type emmodel_options: dict or a list of dict. In the latter case, the size of the list must be the same as the number of layers in the snowpack. :param rtsolver_options: extra to use to create the rtsolver instance (see __init__ of the solver used). :type rtsolver_options: dict

Returns a model instance

get_emmodel(*emmodel*)
get a new emmodel class from the file name

make_emmodel(*emmodel*, *sensor*, *layer*, ***emmodel_options*)
create a new emmodel instance based on the emmodel class or string :param emmodel: type of emmodel to use. Can be given by the name of a file/module in the emmodel directory (as a string) or a class. :param sensor: sensor to use for the calculation. :param layer: layer to use for the calculation

class Model(*emmodel*, *rtsolver*, *emmodel_options=None*, *rtsolver_options=None*)
Bases: `object`

This class drives the whole calculation

set_rtsolver_options(*options=None*, ***kwargs*)
set the option for the rtsolver

set_emmodel_options(*options=None*, ***kwargs*)
set the options for the emmodel

run(*sensor*, *snowpack*, *atmosphere=None*, *snowpack_dimension=None*, *snowpack_column='snowpack'*, *progressbar=False*, *parallel_computation=False*, *runner=None*)
Run the model for the given sensor configuration and return the results

Parameters

- **sensor** – sensor to use for the calculation
- **snowpack** – snowpack to use for the calculation. Can be a single snowpack, a list of snowpack, a dict of snowpack or a SensitivityStudy object.
- **snowpack_dimension** – name and values (as a tuple) of the dimension to create for the results when a list of snowpack is provided. E.g. time, point, longitude, latitude. By default the dimension is called 'snowpack' and the values are rom 1 to the number of snowpacks.
- **snowpack_column** – when snowpack is a DataFrame this argument is used to specify which column contians the Snowpack objects
- **progressbar** – if True, display a progress bar during multi-snowpacks computation
- **parallel_computation** – if True, use the joblib library to run the simulation in parallel. Otherwise, the simulations are run sequentially. See 'runner' arguments.
- **runner** – a 'runner' is a function (or more likely a class with a __call__ method) that takes a function and a list/generator of simulations, executes the function on each simulation and returns a list of results. 'parallel_computation' allows to select between two default (basic) runners (sequential and joblib). Use 'runner' for more advanced parallel distributed computations.

Returns result of the calculation(s) as a `Results` instance

prepare_simulations(*sensor*, *snowpack*, *snowpack_dimension*, *snowpack_column*)

prepare_emmodels(*sensor*, *snowpack*)

run_single_simulation(*simulation*, *atmosphere*)

run_later(*sensor*, *snowpack*, ***kwargs*)

class SequentialRunner(*progressbar=False*)
Bases: `object`

Run the simulations sequentially on a single (local) core. This is the most simple way to run smrt simulations, but the efficiency is poor.

class JoblibParallelRunner(*backend='loky'*, *n_jobs=-1*, *max_numerical_threads=1*)
Bases: `object`

Run the simulations on the local machine on all the cores, using the joblib library for parallelism.

9.11 smrt.core.optional_numba module

9.12 smrt.core.plugin module

register_package(*pkg*)

import_class
Import the modulename and return either the class named "classname" or the first class defined in the module if classname is None.

Parameters

- **scope** – scope where to search for the module.
- **modulename** – name of the module to load.
- **classname** – name of the class to read from the module.

do_import_class(*modulename*, *classname*)

9.13 smrt.core.progressbar module

A progress bar copied and adapted from pyMC code (dec 2014)

class TextProgressBar(*iterations*, *printer*, *width=40*, *interval=None*)
Bases: `smrt.core.progressbar.ProgressBar`

Use *Progress*

animate(*i*, *dummy=None*)

progbar(*i*)

bar(*percent*)

progress_bar(*iters*, *interval=None*)
A progress bar for Python/IPython/IPython notebook

Parameters

- **iters** (*int*) – number of iterations (steps in the loop)
- **interval** – number of intervals to use to update the progress bar (20 by default)

```
from easydev import progress_bar
pb = progress_bar(10)
for i in range(1,10):
    import time
    time.sleep(0.1)
    pb.animate(i)
```

class Progress(*iters*, *interval=None*)

Bases: `object`

Generic progress bar for python, IPython, IPython notebook

```
from easydev import Progress
pb = Progress(100, interval=1)
pb.animate(10)
```

animate(*i*)

elapsed

9.14 smrt.core.result module

The results of RT Solver are hold by the *Result* class. This class provides several functions to access to the Stokes Vector and Muller matrix in a simple way. Most notable ones are `Result.TbV()` and `Result.TbH()` for the passive mode calculations and `Result.sigmaHH()` and `Result.sigmaVV()`. `Result.to_dataframe()` is also very convenient for the sensors with a channel map (all specific satellite sensors have such a map, only generic sensors as `smrt.sensor_list.active()` and `smrt.sensor_list.passive()` does not provide a map by default).

In addition, the RT Solver stores some information in Result.other_data. Currently this includes the effective_permittivity, ks and ka for each layer. The data are accessed directly with e.g. result.other_data['ks'].

To save results of calculations in a file, simply use the pickle module or other serialization schemes. We may provide a unified and inter-operable solution in the future.

Under the hood, *Result* uses xarray module which provides multi-dimensional array with explicit, named, dimensions. Here the common dimensions are frequency, polarization, polarization_inc, theta_inc, theta, and phi. They are created by the RT Solver. The interest

> of using named dimension is that slice of the xarray (i.e. results) can be selected based on the dimension name whereas with numpy the order of the dimensions matters. Because this is very convenient, users may be interested in adding other dimensions specific to their context such
>
> > as time, longitude, latitude, points, ... To do so, *smrt.core.model.Model.run()* accepts a list of snowpack and optionally the parameter *snowpack_dimension* is used to specify the name and values of the new dimension to build.

Example:

```
times = [datetime(2012, 1, 1), datetime(2012, 1, 5), , datetime(2012, 1, 10)]
snowpacks = [snowpack_1jan, snowpack_5jan, snowpack_10jan]

res = model.run(sensor, snowpacks, snowpack_dimension=('time', times))
```

The *res* variable is a *Result* instance, so that for all the methods of this class that can be called, they will return a timeseries. For instance result.TbV(theta=53) returns a time-series of brightness temperature at V polarization and 53° incidence angle and the following code plots this timeseries:

```
plot(times, result.TbV(theta=53))
```

open_result(*filename*)

read a result save to disk. See *Result.save()* method.

make_result(*sensor*, **args*, ***kwargs*)

create an active or passive result object according to the mode

class Result(*intensity*, *coords=None*, *channel_map=None*, *other_data={}*, *mother_df=None*)
Bases: object

Contains the results of a/many computations and provides convenience functions to access these results

coords
Return the coordinates of the result (theta, frequency, ...). Note that the coordinates are also result attribute, so result.frequency works (and so on for all the coordinates).

save(*filename*)
save a result to disk. Under the hood, this is a netCDF file produced by xarray (http://xarray.pydata.org/en/stable/io.html).

sel_data(*channel=None*, ***kwargs*)

return_as_dataframe(*name*, *channel_axis=None*, ***kwargs*)

to_series(***kwargs*)
return the result as a series with the channels defined in the sensor as index. This requires that the sensor has declared a channel list.

class PassiveResult(*intensity*, *coords=None*, *channel_map=None*, *other_data={}*, *mother_df=None*)
Bases: *smrt.core.result.Result*

mode = 'P'

sel_data(*channel=None*, ***kwargs*)

Tb(*channel=None*, ***kwargs*)
Return brightness temperature. Any parameter can be added to slice the results (e.g. frequency=37e9 or polarization='V'). See xarray slicing with sel method (to document). It is also posisble to select by channel if the sensor has a channel_map.

Parameters

- **channel** – channel to select
- ****kwargs** – any parameter to slice the results.

Tb_as_dataframe(*channel_axis=None*, ***kwargs*)
See *PassiveResult()*.to_dataframe

to_dataframe(*channel_axis='auto'*, ***kwargs*)
Return brightness temperature as a pandas.DataFrame. Any parameter can be added to slice the results (e.g. frequency=37e9 or polarization='V'). See xarray slicing with sel method (to document). In addition channel_axis controls the format of the output. If set to None, the DataFrame has a multi-index with all the dimensions (frequency, polarization, ...). If channel_axis is set to "column", and if the sensor has a channel map, the channels are in columns and the other dimensions are in index. If set to "index", the channel are in index with all the other dimensions.

The most conviennent is probably channel_axis="column" while channel_axis=None (default) contains all the data even those not corresponding to a channel and applies to any sensor even those without channel_map. If set to "auto", the channel_axis is "column" if the channel map exit, otherwise is None.

Parameters **channel_axis** – controls whether to use the sensor channel or not and if yes, as a column or index.

TbV(***kwargs*)
Return V polarization. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

TbH (***kwargs*)
Return H polarization. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

polarization_ratio (*ratio='H_V'*, ***kwargs*)
Return polarization ratio. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

class ActiveResult (*intensity*, *coords=None*, *channel_map=None*, *other_data={}*, *mother_df=None*)
Bases: *smrt.core.result.Result*

mode = 'A'

sel_data (*channel=None*, *return_backscatter=False*, ***kwargs*)

sigma (*channel=None*, ***kwargs*)
Return backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9 or polarization='V'). See xarray slicing with sel method (to document). It is also posisble to select by channel if the sensor has a channel_map.

Parameters

- **channel** – channel to select
- ****kwargs** – any parameter to slice the results.

sigma_dB (*channel=None*, ***kwargs*)
Return backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9, polarization_inc='V', polarization='V'). See xarray slicing with sel method (to document)

sigma_as_dataframe (*channel_axis=None*, ***kwargs*)
Return backscattering coefficient as a pandas.DataFrame. Any parameter can be added to slice the results (e.g. frequency=37e9 or polarization='V'). See xarray slicing with sel method (to document). In addition channel_axis controls the format of the output. If set to None, the DataFrame has a multi-index formed with all the dimensions (frequency, polarization, ...). If channel_axis is set to "column", and if the sensor has named channels (channel_map in SMRT wording), the channel are in columns and the other dimensions are in index. If set to "index", the channel are in index with all the other dimensions.

The most conviennent is probably channel_axis="column" while channel_axis=None (default) contains all the data even those not corresponding to a channel and applies to any sensor even those without channel_map.

Parameters **channel_axis** – controls whether to use the sensor channel or not and if yes, as a column or index.

sigma_dB_as_dataframe (*channel_axis=None*, ***kwargs*)
See *ActiveResult()*.to_dataframe

to_dataframe (*channel_axis=None*, ***kwargs*)
Return backscattering coefficient in dB as a pandas.DataFrame. Any parameter can be added to slice the results (e.g. frequency=37e9 or polarization='V'). See xarray slicing with sel method (to document). In addition channel_axis controls the format of the output. If set to None, the DataFrame has a multi-index with all the dimensions (frequency, polarization, ...). If channel_axis is set to "column", and if the sensor has named channels (channel_map in SMRT wording), the channel are in columns and the other dimensions are in index. If set to "index", the channel are in index with all the other dimensions.

If channel_axis is set to "column", and if the sensor has a channel map, the channels are in columns and the other dimensions are in index. If set to "index", the channel are in index with all the other dimensions.

The most conviennent is probably channel_axis="column" while channel_axis=None (default) contains all the data even those not corresponding to a channel and applies to any sensor even those without channel_map. If set to "auto", the channel_axis is "column" if the channel map exit, otherwise is None.

Parameters **channel_axis** – controls whether to use the sensor channel or not and if yes, as a column or index.

to_series(***kwargs*)
return backscattering coefficients in dB as a series with the channels defined in the sensor as index. This requires that the sensor has declared a channel list.

sigmaVV(***kwargs*)
Return VV backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaVV_dB(***kwargs*)
Return VV backscattering coefficient in dB. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaHH(***kwargs*)
Return HH backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaHH_dB(***kwargs*)
Return HH backscattering coefficient in dB. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaHV(***kwargs*)
Return HV backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaHV_dB(***kwargs*)
Return HV backscattering coefficient in dB. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaVH(***kwargs*)
Return VH backscattering coefficient. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

sigmaVH_dB(***kwargs*)
Return VH backscattering coefficient in dB. Any parameter can be added to slice the results (e.g. frequency=37e9). See xarray slicing with sel method (to document)

concat_results(*result_list*, *coord*)
Concatenate several results from *smrt.core.model.Model.run()* (of type *Result*) into a single result (of type *Result*). This extends the number of dimension in the xarray hold by the instance. The new dimension is specified with coord

Parameters

- **result_list** – list of results returned by *smrt.core.model.Model.run()* or other functions.
- **coord** – a tuple (dimension_name, dimension_values) for the new dimension. Dimension_values must be a sequence or

array with the same length as result_list.

Returns *Result* instance

9.15 smrt.core.run_promise module

honour_all_promises(*directory_or_filename*, *save_result_to=None*, *show_progress=True*, *force_compute=True*)

Honour many promises and save the results

Parameters

- **directory_or_filename** – can be a directory, a filename or a list of them
- **save_result_to** – directory where to save the results. If None, the results are not saved. The results are always returned as a list by this function.
- **show_progress** – print progress of the calculation.
- **force_computate** – If False and if a result or lock file is present, the computation is skipped. The order of promise processing is randomized to allow more efficient parallel computation using many calls of this function on the same directory. A lock file is used between the start of a computation and writting the result in order to prevent from running several times the same computation. If the process is interupted (e.g. walltime on clusters), the lock file may persist and prevent any future computation. In this case, lock files must be manually deleted. IF False, the *save_result_to* argument must be set to a valid directory where the results.

honour_promise(*filename*, *save_result_to=None*, *force_compute=True*)

Honour a promise and optionally save the result.

Parameters

- **filename** – file name of the promise
- **save_result_to** – directory where to save the result.
- **force_compute** – see *honour_all_promise*.

load_promise(*filename*)

class RunPromise(*model*, *sensor*, *snowpack*, *kwargs*)

Bases: `object`

run()

save(*directory=None*, *filename=None*)

9.16 smrt.core.sensitivity_study module

SensitivityStudy is used to easily conduct sensitivity studies.

Example:

```
times = [datetime(2012, 1, 1), datetime(2012, 1, 5), , datetime(2012, 1, 10)]
snowpacks = SensitivityStudy("time", times, [snowpack_1jan, snowpack_5jan, snowpack_
↪10jan])


res = model.run(sensor, snowpacks)
```

The *res* variable is a `Result` instance, so that for all the methods of this class that can be called, they will return a timeseries. For instance result.TbV(theta=53) returns a time-series of brightness temperature at V polarization and 53° incidence angle and the following code plots this timeseries:

```
plot(times, result.TbV(theta=53))
```

class SensitivityStudy(*name*, *values*, *snowpacks*)
Bases: object

sensitivity_study(*name*, *values*, *snowpacks*)
create a sensitivity study

Parameters

- **name** – name of the variable to investigate
- **values** – values taken by the variable
- **snowpacks** – list of snowpacks. Can be a sequence or a function that takes one argument and return a snowpack.

In the latter case, the function is called for each values to build the list of snowpacks

9.17 smrt.core.sensor module

The sensor configuration includes all the information describing the sensor viewing geometry (incidence, ...) and operating parameters (frequency, polarization, ...). The easiest and recommended way to create a *Sensor* instance is to use one of the convenience functions such as *passive()*, *active()*, *amsre()*, etc. Adding a function for a new or unlisted sensor can be done in *sensor_list* if the sensor is common and of general interest. Otherwise, we recommend to add these functions in your own files (outside of smrt directories).

passive(*frequency*, *theta*, *polarization=None*, *channel_map=None*, *name=None*)
Generic configuration for passive microwave sensor.

Return a *Sensor* for a microwave radiometer with given frequency, incidence angle and polarization

Parameters

- **frequency** – frequency in Hz
- **theta** – viewing angle or list of viewing angles in degrees from vertical. Note that some RT solvers compute all viewing angles whatever this configuration because it is internally needed part of the multiple scattering calculation. It it therefore often more efficient to call the model once with many viewing angles instead of calling it many times with a single angle.
- **polarization** (*list of characters*) – H and/or V polarizations. Both polarizations is the default. Note that most RT solvers compute all the polarizations whatever this configuration because the polarizations are coupled in the RT equation.
- **channel_map** (*dict*) – map channel names (keys) to configuration (values). A configuration is a dict with frequency, polarization and other such parameters to be used by Result to select the results.
- **name** (*string*) – name of the sensor

Returns *Sensor* instance

Usage example:

```
from smrt import sensor
radiometer = sensor.passive(18e9, 50)
radiometer = sensor.passive(18e9, 50, "V")
radiometer = sensor.passive([18e9,36.5e9], [50,55], ["V","H"])
```

channel_map_for_radar(*frequency=None*, *polarization='HV'*, *order='fp'*)
return a channel_map to convert channel name to frequency and polarization. This function assumes the frequency is coded as a two-digit number in GHz with leading 0 if necessary. The polarization is after the frequency if order is 'fp' and before if order is 'pf'.

active(*frequency*, *theta_inc*, *theta=None*, *phi=None*, *polarization_inc=None*, *polarization=None*, *channel_map=None*, *name=None*)
Configuration for active microwave sensor.

Return a *Sensor* for a radar with given frequency, incidence and viewing angles and polarization

If polarizations are not specified, quad-pol is the default (VV, VH, HV and HH). If the angle of incident radiation is not specified, *backscatter* will be simulated

Parameters

- **frequency** – frequency in Hz
- **theta_inc** – incident angle in degrees from the vertical
- **theta** – viewing zenith angle in degrees from the vertical. By default, it is equal to theta_inc which corresponds to the backscatter direction
- **phi** – viewing azimuth angle in degrees from the incident direction. By default, it is pi which corresponds to the backscatter direction
- **polarization_inc** (*list of 1-character strings*) – list of polarizations of the incidence wave ('H' or 'V' or both.)
- **polarization** (*list of 1-character strings*) – list of viewing polarizations ('H' or 'V' or both)
- **channel_map** (*dict*) – map channel names (keys) to configuration (values). A configuration is a dict with frequency, polarization and other such parameters to be used by Result to select the results.
- **name** (*string*) – name of the sensor

Returns *Sensor* instance

Usage example:

```
from smrt import sensor
scatterometer = sensor.active(frequency=18e9, theta_inc=50)
scatterometer = sensor.active(18e9, 50, 50, 0, "V", "V")
scatterometer = sensor.active([18e9,36.5e9], theta=50, theta_inc=50, polarization_
↪inc=["V", "H"], polarization=["V", "H"])
```

altimeter(*channel*, ***kwargs*)

make_multi_channel_altimeter(*config*, *channel*)

class SensorBase
Bases: object

class Sensor(*frequency=None*, *theta_inc_deg=None*, *theta_deg=None*, *phi_deg=None*, *polarization_inc=None*, *polarization=None*, *channel_map=None*, *name=None*, *wavelength=None*)
Bases: *smrt.core.sensor.SensorBase*

Configuration for sensor. Use of the functions *passive()*, *active()*, or the sensor specific functions e.g. amsre() are recommended to access this class.

wavelength

wavenumber

mode
returns the mode of observation: "A" for active or "P" for passive.

basic_checks()

configurations()

iterate(*axis*)
Iterate over the configuration for the given axis.

Parameters **axis** – one of the attribute of the sensor (frequency, ...) to iterate along

class SensorList(*sensor_list*, *axis='channel'*)
Bases: *smrt.core.sensor.SensorBase*

channel

frequency

configurations()

iterate(*axis=None*)

class Altimeter(*frequency*, *altitude*, *beamwidth*, *pulse_bandwidth*, *sigma_p=None*, *off_nadir_angle=0*, *beam_asymmetry=0*, *ngate=1024*, *nominal_gate=40*, *theta_inc_deg=0.0*, *polarization_inc=None*, *polarization=None*, *channel=None*)
Bases: *smrt.core.sensor.Sensor*

Configuration for altimeter. Use of the functions *altimeter()*, or the sensor specific functions e.g. envisat_ra2() are recommended to access this class.

9.18 smrt.core.snowpack module

Snowpack instance contains the description of the snowpack, including a list of layers and interfaces between the layers, and the substrate (soil, ice, ...).

To create a snowpack, it is recommended to use the *make_snowpack()* function which avoids the complexity of creating each layer and then the snowpack from the layers. For more complex media (like lake ice or sea ice), it may be necessary to directly call the functions to create the different layers (such as *make_snow_layer()*).

Example:

```
# create a 10-m thick snowpack with a single layer,
# density is 350 kg/m3. The exponential autocorrelation function is
# used to describe the snow and the "size" parameter is therefore
# the correlation length which is given as an optional
# argument of this function (but is required in practice)

sp = make_snowpack([10], "exponential", [350], corr_length=[3e-3])
```

class Snowpack(*layers=None*, *interfaces=None*, *substrate=None*, *atmosphere=None*)
Bases: object

holds the description of the snowpack, including the layers, interfaces, and the substrate

nlayer
return the number of layers

layer_thicknesses
return the thickness of each layer

layer_depths
return the depth of the bottom of each layer

bottom_layer_depths
return the depth of the bottom of each layer

top_layer_depths
return the depth of the bottom of each layer

mid_layer_depths
return the depth of the bottom of each layer

z
return the depth of each interface, that is, 0 and the depth of the bottom of each layer

layer_densities
return the density of each layer

profile(*property_name*, *where='all'*, *raise_attributeerror=False*)
return the vertical profile of property_name. The property is searched either in the layer, microstructure or interface.

Parameters

- **property_name** – name of the property
- **where** – where to search the property. Can be 'all', 'layer', 'microstructure', or 'interface'
- **raise_attributeerror** – raise an attribute error if the attribute is not found

append(*layer*, *interface=None*)
append a new layer at the bottom of the stack of layers. The interface is that at the top of the appended layer.

Parameters

- **layer** – instance of `Layer`
- **interface** – type of interface. By default, flat surface (`Flat`) is considered meaning the coefficients are calculated with Fresnel coefficient and using the effective permittivity of the surrounding layers

delete(*ilayer*)
delete a layer and the upper interface

Parameters **ilayer** – index of the layer

copy()
make a shallow copy of a snowpack by copying the list of layers and interfaces but not the layers and interfaces themselves which are still shared with the original snowpack. This method allows the user to create a new snowpack and remove, append or replace some layers or interfaces afterward. It does not allow to alter the layers or interfaces without changing the original snowpack. See py:meth:~deepcopy.

deepcopy()
make a deep copy of a snowpack.

basic_check()

check_addition_validity(*other*)

update_layer_number()

to_dataframe(*default_columns=True*, *other_columns=None*)

9.19 smrt.core.test_globalconstants module

test_density_of_ice()

test_freezing_point()

test_permittivity_of_air()

test_speed_of_light()

9.20 smrt.core.test_interface module

test_make_interface_noargs()

9.21 smrt.core.test_layer module

test_microstructure_model()

9.22 smrt.core.test_lib module

setup_func_sp()

setup_func_em(*testpack=None*)

test_generic_ft_even_matrix()

9.23 smrt.core.test_result module

test_methods()

test_positive_sigmaVV()

test_positive_sigmaVH()

test_positive_sigmaHV()

test_positive_sigmaHH()

test_sigma_dB()

test_sigma_dB_as_dataframe()

test_to_dataframe_wtih_channel_axis_on_column()

test_to_dataframe_wtihout_channel_axis()

test_return_as_series()

test_concat_results()

test_concat_results_other_data()

9.24 smrt.core.test_sensor module

test_iterate()

test_wavelength()

test_no_theta()

test_passive_wrong_frequency_units_warning()

test_duplicate_theta()

test_duplicate_theta_active()

test_passive_mode()

test_active_wrong_frequency_units_warning()

test_active_mode()

9.25 smrt.core.test_snowpack module

test_profile()

create_two_snowpacks()

test_addition()

test_layer_addition()

test_inplace_addition()

test_inplace_layer_addition()

test_substrate_addition()

test_atmosphere_addition()

test_atmosphere_addition_double_snowpack()

test_invalid_addition_atmosphere()

test_invalid_addition_atmosphere2()

test_invalid_addition_substrate()

test_invalid_addition_substrate2()

9.26 Module contents

The *core* package contains the SMRT machinery. It provides the infrastructure that provides basic objects and orchestrates the "science" modules in the other packages (such as *smrt.emmodel* or *smrt.rtsolver*).

Amongst all, we suggest looking at the documentation of the *Result* object.

For developers

We strongly warn against changing anything in this directory. In principle this is not needed because no "science" is present and most objects and functions are generic enough to be extendable from outside (without affecting the core definition). Ask advice from the authors if you really want to change something here.

CHAPTER 10

smrt.utils package

10.1 Submodules

10.2 smrt.utils.dmrt_qms_legacy module

Wrapper to the original DMRT_QMS matlab code using the SMRT framework. To use this module, extra installation are needed:

- get DMRT_QMS from http://web.eecs.umich.edu/~leutsang/Available%20Resources.html and extract the model somewhere
- install the oct2py module using `pip install oct2py` or `easy_install install oct2py`
- install Octave version 3.6 or above.
- for convenience you can set the DMRT_QMS_DIR environment variable to point to DMRT-QMS path. This path can also be programmatically set with and use *`set_dmrt_qms_path()`* function.

In case of problem check the instructions given in http://blink1073.github.io/oct2py/source/installation.html

You may also want to increase the number of streams in passive/DMRT_QMS_passive.m

`set_dmrt_qms_path`(*path*)
set the path where dmrt_qms archive has been uncompressed, i.e. where the file *dmrt_qmsmain.m* is located.

`run`(*sensor*, *snowpack*, *dmrt_qms_path=None*, *snowpack_dimension=None*, *full_output=False*)
call DMRT-QMS for the snowpack and sensor configuration given as argument. The *`sticky_hard_spheres`* microstructure model must be used.

Parameters

- **`snowpack`** – describe the snowpack.
- **`sensor`** – describe the sensor configuration.
- **`full_output`** – determine if ks, ka and effective permittivity are return in addition to the result object

dmrt_qms_active(*sensor*, *snowpack*)

dmrt_qms_emmodel(*sensor*, *layer*, *dmrt_qms_path=None*)
Compute scattering and absorption coefficients using DMRT QMS

Parameters

- **layer** – describe the layer.
- **sensor** – describe the sensor configuration.

10.3 smrt.utils.hut_legacy module

Wrapper to original HUT matlab using SMRT framework. To use this module, extra installation are needed:

- get HUT. Decompress the archive somewhere on your disk.
- in the file snowemis_nlayers change the 6 occurences of the "do" variable into "dos" because it causes a syntax error in Octave.
- install the oct2py module using `pip install oct2py` or `easy_install install oct2py`.
- install Octave version 3.6 or above.
- for convenience you can set the HUT_DIR environment variable to point to HUT path. This path can also be programmatically set with *set_hut_path()*.

In case of problem check the instructions given in http://blink1073.github.io/oct2py/source/installation.html

set_hut_path(*path*)
set the path where MEMLS archive has been uncompressed, i.e. where the file *memlsmain.m* is located.

run(*sensor*, *snowpack*, *ke_option=0*, *grainsize_option=1*, *hut_path=None*)
call HUT for the snowpack and sensor configuration given as argument. Any microstructure model that defines the "radius" parameter is valid.

Parameters

- **snowpack** – describe the snowpack.
- **sensor** – describe the sensor configuration.
- **ke_option** – see HUT snowemis_nlayers.m code
- **grainsize_option** – see HUT snowemis_nlayers.m code

10.4 smrt.utils.memls_legacy module

Wrapper to the original MEMLS matlab code using the SMRT framework. To use this module, extra installation are needed:

- download MEMLS from http://www.iapmw.unibe.ch/research/projects/snowtools/memls.html. Decompress the archive somewhere on your disk.
- install the oct2py module using `pip install oct2py` or `easy_install install oct2py`
- install Octave version 3.6 or above.
- for convenience you can set the MEMLS_DIR environment variable to point to MEMLS path. This path can also be programmatically set with *set_memls_path()*

In case of problem check the instructions given in http://blink1073.github.io/oct2py/source/installation.html

set_memls_path(*path*)
set the path where MEMLS archive has been uncompressed, i.e. where the file *memlsmain.m* is located.

run(*sensor*, *snowpack*, *scattering_choice=12*, *atmosphere=None*, *memls_path=None*, *memls_driver=None*, *snowpack_dimension=None*)
call MEMLS for the snowpack and sensor configuration given as argument. Any microstructure model that defines the "corr_length" parameter is valid, but it must be clear that MEMLS only considers exponential autocorrelation.

Parameters

- **snowpack** – describe the snowpack.
- **sensor** – describe the sensor configuration.
- **scattering_choice** – MEMLS proposes several formulation to compute scattering_function. scattering_choice=ABORN (equals 12) is the default here and is recommended choice to compare with IBA. Note that some comments in memlsmain.m suggest to use scattering_choice=MEMLS_RECOMMENDED (equals 11). Note also that the default grain type in memlsmain is graintype=1 corresponding to oblate spheroidal calculation of effective permittivity from the empirical representation of depolarization factors. To use a Polder-Van Santen representation of effective permittivity for small spheres, graintype=2 must be set in your local copy of MEMLS.
- **atmosphere** – describe the atmosphere. Only tbdown is used for the Tsky argument of memlsmain.
- **memls_path** – directory path to the memls Matlab scripts
- **memls_driver** – matlab function to call to run memls. memlsmain.m is the default driver in the original MEMLS distribution for the passive case and amemlsmain.m for the active case.
- **snowpack_dimension** – name and values (as a tuple) of the dimension to create for the results when a list of snowpack is provided. E.g. time, point, longitude, latitude. By default the dimension is called 'snowpack' and the values are from 1 to the number of snowpacks.

memls_emmodel(*sensor*, *layer*, *scattering_choice=12*, *graintype=2*)
Compute scattering (gs6) and absorption coefficients (gai) using MEMLS

Parameters

- **layer** – describe the layer.
- **sensor** – describe the sensor configuration.
- **scattering_choice** – MEMLS proposes several formulation to compute scattering_function. scattering_choice=ABORN (equals 12) is the defaut here and is recommended choice to compare with IBA.

10.5 smrt.utils.mpl_plots module

plot_snowpack(*sp*, *show_vars=None*, *show_shade=False*, *ax=None*)

plot_streams(*sp*, *emmodel*, *sensor*, *ilayer=None*, *ax=None*)

format_vars(*lay*, *show_vars*, *delimiter=' '*)

class CosineComputor
Bases: `object`

solve(*snowpack*, *emmodel_instances*, *sensor*, *atmosphere*)

class ReciprocalScale(*axis*)

Bases: matplotlib.scale.LinearScale

name = 'stickiness_reciprocal'

set_default_locators_and_formatters(*axis*)

Set the locators and formatters of *axis* to instances suitable for this scale.

get_transform()

Return the transform for linear scaling, which is just the *~matplotlib.transforms.IdentityTransform*.

class ReciprocalTransform(*shorthand_name=None*)

Bases: matplotlib.transforms.Transform

input_dims = 1

output_dims = 1

is_separable = True

transform_non_affine(*a*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

values [array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

array The output values as NumPy array of length *output_dims* or shape (N x *output_dims*), depending on the input.

inverted()

Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

has_inverse = True

class InvertedReciprocalTransform(*shorthand_name=None*)

Bases: matplotlib.transforms.Transform

input_dims = 1

output_dims = 1

is_separable = True

transform_non_affine(*a*)

Apply only the non-affine part of this transformation.

`transform(values)` is always equivalent to `transform_affine(transform_non_affine(values))`.

In non-affine transformations, this is generally equivalent to `transform(values)`. In affine transformations, this is always a no-op.

values [array] The input values as NumPy array of length *input_dims* or shape (N x *input_dims*).

array The output values as NumPy array of length *output_dims* or shape (N x *output_dims*), depending on the input.

inverted()
Return the corresponding inverse transformation.

It holds `x == self.inverted().transform(self.transform(x))`.

The return value of this method should be treated as temporary. An update to *self* does not cause a corresponding update to its inverted copy.

has_inverse = True

10.6 smrt.utils.repo_tools module

General tools related to code repository

get_hg_rev(*file_path*)
get_hg_rev is a tool to print out which commit of the model you are using.

This is useful when revisiting ipython notebooks, can be used to compare the original model commit ID with the latest version.

Usage:

```
from smrt.utils.repo_tools import get_hg_rev
path_to_file = "/path/to/your/repository"
get_hg_rev(path_to_file)
```

Note: This is for a mercurial repository

10.7 Module contents

This packages contain various utilities that works with/for SMRT.

The wrappers to legacy snow radiative transfer models can be used to run DMRT-QMS (passive mode), HUT and MEMLS (passive mode). Other tools are listed below.

dB(*x*)
computes the ratio x in dB.

invdB(*x*)
computes the dB value x in natural value.

CHAPTER 11

Guidelines for Developers

At the moment this is an organic document to collect all the model design and developer style decisions. This will also include information on how to get started with useful developer tools. At the moment, it contains personal experience of installing and using these tools although these may be removed if they do not appear to be useful to others.

These guidelines will be turned into a formal document towards the end of the project.

11.1 Use of import statements

Good rules for python imports

In short:

- use fully qualified names
- `from blabla import *` should never be used.
- `from blabla import passive` should be avoided in SMRT but can be used in user code.
- keep at least the module e.g. “from smrt import sensor_list” is the best compromise.
- use “as” with moderation and everyone should agree to use it.
- but `import numpy as np` is good.
- to start, we will use an explicit import at the top of the driver file, making the code more cumbersome, but may later consider a plugin framework to do the import and introspection in a nice way.

Note: it’s part of the Google Python style guides that all imports must import a module, not a class or function from that module. There are way more classes and functions than there are modules, so recalling where a particular thing comes from is much easier if it is prefixed with a module name. Often multiple modules happen to define things with the same name – so a reader of the code doesn’t have to go back to the top of the file to see from which module a given name is imported.

11.2 Python

Python was chosen because of its growing use in the scientific community and higher flexibility than compiled legacy languages like FORTRAN. This enables the model to be modular much more easily, which is a main constraint of the project, allows faster development and an easier exploration of new ideas. The performance should not be an issue as the time consuming part of the model should be localized in the RT solver and numerical integrations which uses the highly optimized scipy module facility that basically uses BLAS, LAPACK and MINPACK libraries as would be done in FORTRAN. Compilation of the Python code with Numba or Pypy will be considered in case of performance issues later in the project or even more probably after. Parallelization could be done later e.g. through joblib module.

The model in the framework of the current project mainly aims at exploring new ideas involving the microstructure and tests various modelling solutions. It is quite likely that operational needs (especially very intensive ones) will require rewritting a selected subset of the model.

11.2.1 Python versions

The target version is Python 3.4+ which is better optimized and is the only supported version in the future (after 2020) with the use of a subset syntax to ensure compatibility with the lastest 2.7.x and PyPy. It means in practice that the model will be compatible with the last 2.7.x version but is "ready" for Python 3 and later. For this "__future__" directives and six module will be used. The tests must pass the two versions. This choice is overall a weak constraint for developers and big asset for users.

Anaconda is probably the easiest way to install python, especially when several versions are needed. See also Installing multiple versions of python is system dependent and also depends on your preferred install method.

Perhaps it's not strictly necessary to follow all steps, but I followed these instructions for Mac OSX to install python 3.5. Then `pip` installs packages into python 2.7 and `pip3` installs packages into python 3.5. Note on Tcl/Tk for Mac OSX. I have installed ActiveTcl 8.6.4 and am keeping my fingers crossed that these changes have not broken anything…I have subsequently installed python 3.4.3. This means that `python3` will run version 3.4.3 by default. It doesn't seem trivial to get `python3` to point back to python 3.5, but that's probably ok as the target version is 3.4, and it will be worth testing for 3.5 alongside.

11.2.2 tox: testing multiple python versions

The tox package allows multiple versions of python to be tested. Although not clear whether this needs to be installed in python 2 or 3, I installed with `pip` rather than `pip3` and trust that it will take care of everything. This seems to work fine.

The setup to run tox is contained in the tox.ini file. At the moment this is setup for nosetests against python versions 2.7, 3.4 and 3.5. Also, at present tox.ini does not require a setup.py to run. Once the model is fully operational the line `skipsdist = True` should be deleted, or this parameter set to False. Note that all modules to be imported need to be listed in the dependencies (deps) in the tox.ini file. An ImportError may indicate that the module it is trying to import has not been included in the tox.ini.

To run the nosetests for all the different versions, using the installed tox package, simply type:

```
tox
```

If you want to test for only one python version, type e.g:

```
tox -e py27
```

11.3 setup.py

This is needed in order to build, install and distribute the model through Distutils (instructions). To be done for the public release.

11.4 bug correction

Every bug should result in writing a test.

11.5 Classes

If the compulsary argument list becomes too long (say 4?), use optional arguments to make things easier to read.

Guidelines on number of parameters a function should take.

Merge two objects in python.

11.6 PEP008

Code must conform to PEP8 - with the exception that lines of up to 140 characters are allowed and extra space are allowed in long formula for readability. Particular points of note:

- 4 spaces for the indentation.
- one space after comma and around operators.
- all names (variable, function, ...) are meaningful. Abbreviations are used in a very limited number of cases.
- function names are lowercase only and word a spaced by underscore.
- Constants are usually defined on a module level and written in all capital letters with underscores separating words

You can check for PEP8 compliance automatically with nosetests. To do this, install tissue and pep8. Then type:

```
nosetests --with-tissue --tissue-ignore=E501
```

or:

```
nosetests --with-tissue --tissue-ignore=E501 **specific filename**
```

to run nosetests with the pep8 checks. As we have allowed 140 characters per line, the E501 longer line warning needs to be suppressed.

11.7 Sphinx

Documentation is done in-code, and is automatically generated with Sphinx. If no new modules are added, generate the rst and html documentation from the in-code Sphinx comments, by typing (whilst in smrt/doc directory):

```
make fullhtml
```

The documentation can be accessed via the index.html page in the smrt/doc/build/html folder.

If you have math symbols to be displayed, this can be done with the imgmath extension (already used), which generates a png and inserts the image at the appropriate place. You may need to set the path to latex and dvipng on your system. From the source directory, this can be done with e.g.:

```
sphinx-build -b html -D imgmath_latex=/sw/bin/latex -D imgmath_dvipng=/sw/bin/dvipng .
↪ ../build/html
```

or to continue to use :`make html` or `make fullhtml`, by setting your path (C-shell) e.g.:

```
set path = ($path /sw/bin)
```

or bash:

```
PATH=$PATH:/sw/bin
```

Note: Math symbols will need double backslashes in place of the single backslash used in latex.

To generate a list of undocumented elements, whilst in the *source* directory:

```
sphinx-build -b coverage . coverage
```

The files will be listed in the *coverage/python.txt* file

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V