SMRT Documentation

Release 1.0

G. Picard, M. Sandells, H. Löwe

Sep 11, 2020

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

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.

1.1 Submodules

1.1.1 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_snowpack(*thickness*, *microstructure_model*, *density*, *interface=None*, *substrate=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.
- **microstructure_model** – microstructure_model to use (e.g. sticky_hard_spheres or independent_sphere or exponential)
- **interface** – type of interface, flat/fresnel is the default
- **density** – densities of the layers

All the other parameters (temperature, microstructure parameters, emmodel, etc, 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.

TODO: include the documentation of make_snow_layer here once stabilized

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*, *liquid_water=0*, *salinity=0*, ***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 $^{-3}$
- **temperature** – temperature of layer in K
- **ice_permittivity_model** – permittivity formulation (default is ice_permittivity_matzler87)
- **liquid_water** – volume of liquid water with respect to ice volume (default=0)

- **salinity** – salinity in kg/kg, for using PSU as unit see PSU constant in smrt module (default = 0)
- **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 `Layer` instance

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*, *interface=None*, *substrate=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 is modelled as scattering brines embedded in a pure ice background - Multi year is modelled as scattering air bubbles in a saline ice background (but brines are non-scattering in this case). - Lake 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", "lake"
- **thickness** – thicknesses of the layers in meter (from top to bottom). The last layer thickness can be "numpy.inf" for a semi-infinite layer.
- **temperature** – temperature of ice/water in K
- **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,
- **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.
- **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
- **density** – density of ice layer in kg m $^{-3}$
- **porosity** – porosity of ice layer (in %). Default is 0.
- **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 interface: type of interface, flat/fresnel is the default :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*, ***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 arguments 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)
- **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
- **density** – (multiyear) density of ice layer in kg m $^{-3}$. If not given, density is calculated from temperature, salinity and ice porosity.
- **porosity** – (mutliyear and fresh) air porosity of ice layer (0..1). Default is 0.
- **ice_permittivity_model** – (all) pure ice permittivity formulation (default is ice_permittivity_matzler06)
- **brine_permittivity_model** – (firstyear and multiyear) brine permittivity formulation (default is brine_permittivity_stogryn85)
- **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). See the documentation of the microstructure model.

Returns `Layer` instance

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 -2–30oC. For higher temperatures (>2oC) 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*)

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.
- **temperature** – temperature of layers in K
- **ks** – scattering coefficient of layers in m^-1
- **ka** – absorption coefficient of layers in m^-1
- **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

1.1.2 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" or "dobson85"), a function of frequency and temperature or a complex value.
- **moisture** – soil moisture in m:sup:*3* m:sup:*-3* to compute the permittivity. This parameter is used depending on the permittivity_model.
- **sand** – soil relative sand content. This parameter is used or not depending on the permittivity_model.
- **clay** – soil relative clay content. This parameter is used or not depending on the permittivity_model.
- **drymatter** – soil content in dry matter in kg m:sup:*-3*. This parameter is used or not depending on the permittivity_model.
- ****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*)

1.1.3 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=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.

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=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)

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
```

quickscat(*channel=None*, *theta=None*, *polarization=None*)

Configuration for quickscat sensor.

This function can be used to simulate the 4 QUICKSCAT 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 return 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

decompose_channel(*channel*, *lengths*)

CHAPTER 2

smrt.permittivity package

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()`.

2.1 Submodules

2.1.1 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)

2.1.2 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)
```

2.1.3 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)

2.1.4 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

2.1.5 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.

2.1.6 smrt.permittivity.water module

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

2.1.7 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) :returns: Complex permittivity of pure ice

CHAPTER 3

smrt.microstructure_model package

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.

3.1 Submodules

3.1.1 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.

3.1.2 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 = {}

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

3.1.3 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 = {}

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

3.1.4 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 = []

optional_args = {}

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

3.1.5 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 = {}

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).

3.1.6 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 = {}

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

3.1.7 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}

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

3.1.8 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 = {}

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).

CHAPTER 4

smrt.interface package

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.

4.1 Submodules

4.1.1 smrt.interface.coherent_flat module

4.1.2 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

Bases: `object`

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

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
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

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

4.1.3 smrt.interface.geometrical_optics module

4.1.4 smrt.interface.geometrical_optics_backscatter module

4.1.5 smrt.interface.transparent module

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

class Transparent

Bases: object

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

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

4.1.6 smrt.interface.vector3 module

CHAPTER 5

smrt.substrate package

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.

5.1 Submodules

5.1.1 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*, *smrt.interface.flat.Flat*

emissivity_matrix (*frequency*, *eps_1*, *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

specular_reflection_matrix(*frequency*, *eps_1*, *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
- **mu1** – array of cosine of incident angles
- **npol** – number of polarization

Returns the reflection matrix

5.1.2 smrt.substrate.geometrical_optics module

5.1.3 smrt.substrate.geometrical_optics_backscatter module

5.1.4 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)
```

(continues on next page)

(continued from previous page)

```
# 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*)
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*)

5.1.5 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(*m*, *frequency*, *eps_1*, *mu1*, *npol*)

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

5.1.6 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*)

5.1.7 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*)

CHAPTER 6

smrt.atmosphere package

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.

6.1 Submodules

6.1.1 smrt.atmosphere.simple_isotropic_atmosphere module

Implement an isotropic atmosphere with prescribed emission (up and down) and transmittivity

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

Bases: `object`

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

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

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

CHAPTER 7

smrt.emmodel package

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.

7.1 Submodules

7.1.1 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()

7.1.2 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()

7.1.3 smrt.emmodel.dmrt_shortrange module

7.1.4 smrt.emmodel.effective_permittivity module

This module contains functions that are not specific 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:

```
# If imported to an electromagnetic model:
from .commonfunc import depolarization_factors
depol_xyz = depolarization_factors()

# General import:
from smrt.emmodel.commonfunc import depolarization_factors
depol_xyz = depolarization_factors(length_ratio=1.2)
```

polder_van_santen(*frac_volume*, *e0=None*, *eps=None*, *depol_xyz=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.
- **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 .commonfunc 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=None*, *eps=None*, *depol_xyz=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.
- **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 .commonfunc 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

maxwell_garnett (*frac_volume*, *e0*, *eps*, *depol_xyz=None*, *inclusion_shape=None*)
Calculates effective permittivity of snow by solution of 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

:added **kwargs to enable same structure of calling maxwell_garnett() and polder_van_santen() as effective_permittivity_model... :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
```

7.1.5 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. Must be a function of 4 parameters (frac_volume, e0, es, depol_xyz).

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

class IBA(*sensor*, *layer*)

Bases: object

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=None*, *eps=None*, *depol_xyz=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.
- **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 .commonfunc 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})
```

(continues on next page)

(continued from previous page)

```
# 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()

set_max_mode(*m_max*)
Sets the maximum level of phase matrix Fourier decomposition needed. Called by the radiative transfer solver.

Parameters **m_max** – maximum Fourier decomposition mode needed

Note: m_max = 0 for passive

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.

ft_even_phase(*m*, *mu_s*, *mu_i*, *npol=None*)
IBA phase matrix.

These are the Fourier decomposed phase matrices for modes m = 0, 1, 2.... This method creates or accesses the cache of ft_phase so Fourier Decomposition only needs to be done once per layer for all modes.

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

- **m** – mode for decomposed phase matrix (0, 1, 2)
- **mu** – 1-D array of cosines of incidence angle
- **npol** – [Optional] number of polarizations - normally set from sensor properties

Returns cached_phase[m] cached phase matrix for all scattering streams for one Fourier Decomposition mode

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

IBA Phase function (not decomposed).

precompute_ft_even_phase(*mu*, *m_max*, *npol*)

Calculation of the Fourier decomposed IBA phase function.

This method calculates the Improved Born Approximation phase matrix for all Fourier decomposition modes and stores the output in a cache so the calculation is not repeated for each mode. The radiative transfer solver then accesses the cache.

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

Calculates cached_phase: Stored phase matrix for each Fourier mode m

---

Note: The size of the cached_phase[m] matrix depends on the mode. Only p11, p12, p21 and p22 elements are needed for the m = 0 mode, whereas an extended matrix with the p13, p23, p31, p32 and p33 elements are required for m > 0 modes (active only). The size of the cached phase matrix will also vary with snow layer, as it depends on the number of streams simulated (length of mu).

---

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*)
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

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()
Calculation of complex effective permittivity of the medium.

Returns effective_permittivity complex effective permittivity of the medium

class IBA_MM(*sensor*, *layer*)
Bases: *smrt.emmodel.iba.IBA*

7.1.6 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

7.1.7 smrt.emmodel.nonscattering module

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

class NonScattering(*sensor*, *layer*)
Bases: object

basic_check()

set_max_mode(*m_max*)

ft_even_phase(*m*, *mu_s*, *mu_i*, *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*)

effective_permittivity()

7.1.8 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*

7.1.9 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(*m*, *mu_s*, *mu_i*, *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(*m*, *mu_s*, *mu_i*, *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(*m*, *mu_s*, *mu_i*, *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 = [PCvvp PCvhp -PSvup] [PChvp PChhp -PShup] [PSuvp PSuhp PCuup]

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(*m*, *mu_s*, *mu_i*, *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*)
return the extinction coefficient

effective_permittivity()

7.1.10 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*

CHAPTER 8

smrt.rtsolver package

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.

8.1 Submodules

8.1.1 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_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'*, *phase_normalization=True*, *error_handling='exception'*)

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.

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*, *eigenvalue_solver*, *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*)

class EigenValueSolver(*ke*, *ks*, *ft_even_phase*, *mu*, *weight*, *normalization*)

Bases: `object`

solve(*m*, *compute_coherent_only*)

normalize(*m*, *A*)

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

gaussquad(*n*)

8.1.2 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*)

CHAPTER 9

smrt.core package

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.

9.1 Submodules

9.1.1 smrt.core.check_numba module

9.1.2 smrt.core.error module

Definition of the Exception specific to SMRT.

exception SMRTError
Bases: Exception

Error raised by the model

exception SMRTWarning
Bases: Exception

Warning raised by the model

9.1.3 smrt.core.filelock module

9.1.4 smrt.core.fresnel module

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

fresnel_coefficients(*eps_1*, *eps_2*, *mu1*)
compute the reflection in two polarizations (H and V)

Parameters

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

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

fresnel_reflection_matrix(*eps_1*, *eps_2*, *mu1*, *npol*, *return_as_diagonal=False*)
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*, *return_as_diagonal=False*)
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*

9.1.5 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
```

9.1.6 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 instannce class corresponding to the interface model.

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

Parameters **class_or_modulename** – a class or name of the python module in smrt/interface

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

Abstract class for interface between layer or 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 argument handling and calculation of the permittivity constant for soil case.

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

9.1.7 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 snow 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

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.

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*, ***kwargs*)

This decorator is used for the permittivity functions. It declares the layer properties needed to call the function and the optiona once. This allows permittivity functions to use any properties of the layer, as long as it is defined.

9.1.8 smrt.core.lib module

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

is_sequence(*x*)

class 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 diagional object without numpy subclassing and without much features. It seems that proper subclassing numpy and overloading matmul is a very difficult problem.

as_dia_matrix()

diagonal()

check_type(*other*)

9.1.9 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 *run()* method can be used with list of snowpacks. In this case, it is recommended to set the snowpack_dimension_name and snowpack_dimension_values variable which gives the name and values of the coordinates that are create for the Results. This is useful with timeseries for instance.

Example:

```
snowpacks = []
times = []
for file in filenames:
    #  create a snowpack for each time series
    sp = ...
    snowpacks.append(sp)
    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.

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** (*string or class or list of strings or classes. If a list is given, different models are used for the different layers of the snowpack. In this case, the size of the list must be the same as the number of layers in the snowpack.*) – 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.
- **rtsolver** (*string or class. Can be None when only computation of the layer electromagnetic properties is needed.*) – type of solver to use. Can be given by the name of a file/module in the rtsolver directeory (as a string) or a class.
- **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.*) – extra arguments to use to create emmodel instance. Valid arguments depend on the selected emmodel. It is documented in for each emmodel class.
- **rtsolver_options** (*dict*) – extra to use to create the rtsolver instance (see __init__ of the solver used).

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. :type emmodel: string or class or list of strings or classes. If a list is given, different models are used for the different layers of the snowpack. In this case, the size of the list must be the same as the number of layers in the snowpack. :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*, *progressbar=False*)
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 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 from 1 to the number of snowpacks.
- **progressbar** – if True, display a progress bar during multi-snowpacks computation

Returns result of the calculation(s) as a `Results` instance

run_later(*sensor*, *snowpack*, ***kwargs*)

9.1.10 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*)

9.1.11 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

9.1.12 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()*. Other methods could be developed for specific needs.

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.

class Result(*intensity*, *coords=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).

Tb(***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)

Tb_as_dataframe(***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)

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)

sigma(***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_dB(***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(***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_dB_as_dataframe(***kwargs*)

Return backscattering coefficient in dB. 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)

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)

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).

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

9.1.13 smrt.core.run_promise module

honour_all_promises (*directory_or_filename*, *save_result_to=None*, *show_progress=True*)
Honour many promises and save the results

Parameters **directory_or_filename** – can be a directory, a filename or a list of them

honour_promise (*filename*, *save_result_to=None*)
Honour a promise and optionally save the result

load_promise (*filename*)

class RunPromise (*model*, *sensor*, *snowpack*, *kwargs*)
Bases: `object`

run ()

save (*directory=None*, *filename=None*)

9.1.14 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

9.1.15 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=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.

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=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)

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"])
```

class SensorBase

Bases: object

class Sensor(*frequency=None*, *theta_inc_deg=None*, *theta_deg=None*, *phi_deg=None*, *polarization_inc=None*, *polarization=None*, *channel=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

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

configurations()

iterate(*axis=None*)

9.1.16 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*)

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

z
return the depth of each interface, that is, 0 and the depth of the bottom of each layer

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

basic_check()

check_addition_validity(*other*)

CHAPTER 10

smrt.utils package

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.

10.1 Submodules

10.1.1 smrt.utils.dmrt_qms_legacy module

10.1.2 smrt.utils.hut_legacy module

10.1.3 smrt.utils.memls_legacy module

10.1.4 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*, ***kwargs*)
Bases: `matplotlib.scale.LinearScale`

name = 'stickiness_reciprocal'

set_default_locators_and_formatters(*axis*)
Set the locators and formatters to reasonable defaults for linear scaling.

get_transform()
The transform for linear scaling is just the `IdentityTransform`.

class ReciprocalTransform(*shorthand_name=None*)
Bases: `matplotlib.transforms.Transform`

input_dims = 1

output_dims = 1

is_separable = True

transform_non_affine(*a*)
Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x *input_dims*) and returns a numpy array of shape (N x *output_dims*).

Alternatively, accepts a numpy array of length *input_dims* and returns a numpy array of length *output_dims*.

inverted()
Return the corresponding inverse transformation.

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.

`x === self.inverted().transform(self.transform(x))`

class InvertedReciprocalTransform(*shorthand_name=None*)
Bases: `matplotlib.transforms.Transform`

input_dims = 1

output_dims = 1

is_separable = True

transform_non_affine(*a*)
Performs only the non-affine part of the 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.

Accepts a numpy array of shape (N x *input_dims*) and returns a numpy array of shape (N x *output_dims*).

Alternatively, accepts a numpy array of length *input_dims* and returns a numpy array of length *output_dims*.

inverted()
Return the corresponding inverse transformation.

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.

```
x === self.inverted().transform(self.transform(x))
```

10.1.5 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

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

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

Z