PyARTS User Guide, Algorithm Description and Theoretical Basis

Cory Davis and Gerrit Holl

date: 2011-05-18

PyARTS version: 1.2.3

email: corzneffect@gmail.com, gerrit.holl@ltu.se

Contents

Introduction 3

- About this document . . . 3
- Why Python? . . . 3
- Downloading PyARTS . . . 3
- Prerequisites . . . 3
- Installation . . . 4
- Testing your Installation . . . 4
- Examples . . . 4
- Documentation . . . 4
- History . . . 4
- Acknowledgments . . . 5

PyARTS: an ARTS related Python package 6

- Introduction . . . 6
- An example . . . 7

The arts module 9

- ArtsRun objects . . . 9
- Other module contents . . . 10

The clouds module 11

- Clouds . . . 11
- Cloud objects . . . 11
- Hydrometeor objects . . . 13
- Other module contents . . . 15
- Algorithm Description and Theoretical Basis . . . 16

The arts_scat module 21

- SingleScatteringData objects . . . 21
- Other module contents . . . 22
- Algorithm and Theoretical Basis . . . 23
- The range of convergent size parameters and aspect ratios for ice crystal optical property generation . . . 25

The arts_types module 27

- LatLonGriddedField3 objects . . . 27

The plotting module 28
SentinelMap objects 28
Other module contents 28

The artsXML module 30
artsXML 30
XMLfile objects 30
Other module contents 30

The arts_math module 32
Other module contents 32

The io module 33
IO 33
Other module contents 33

The general module 34
Selected functions 34

The sli module 35
SLIData2 objects 35

Introduction

About this document

This document has two purposes; it aims to serve as a user guide, with descriptions of important functions and classes and examples of their use, and also where necessary, there are algorithm descriptions, and theoretical arguments justifying these algorithms.

Much of the user guide components of this document have been automatically extracted from the *docstrings* in the PyARTS source code (see http://epydoc.sourceforge.net/docstrings.html). This ensures consistency between the user guide and on-line help, keeps the user guide up-to-date, and also improves the quality of the on-line help.

Algorithm description and theoretical basis (ATBD) content has only been included in cases where the algorithm in question is novel and complex. The most mathematically arduous component of this package is the T -matrix code of Mishchenko. Since this code has been included in only a very slightly modified form, there is no T -matrix ATBD information included in this document. Instead the user is directed to the appropriate papers by Mishchenko *et al* (see The arts_scat module for exact references), which are all available from http://www.giss.nasa.gov/~crmim/publications.

Why Python?

The following are some of the reasons that make Python an attractive language for scientific computing

- straightforward incorporation of FORTRAN code
- elegant syntax
- Fully object oriented
- Interactive
- comprehensive standard library
- powerful, and freely available third-party scientific packages (NumPy, SciPy, matplotlib)
- platform independent
- straightforward package distribution
- **FREE** as in beer
- **FREE** as in speech

In the case of PyARTS, the choice of Python was primarily based on the first point, which was very important given the reliance upon pre-existing fortran code, and also the last points, which removes an important obstruction in the sharing of code between scientists. Interactivity, and the availability of high level, well documented libraries, contributed to the rapid development of the PyARTS package.

Downloading PyARTS

I think you have already downloaded PyARTS succesfully, but otherwise, PyARTS is available by svn from the ARTS website http://www.sat.ltu.se/arts/tools.

Prerequisites

Note: those are the versions that PyARTS was most recently developed and tested with. It may work with older versions and probably will work with newer versions (possibly with minor changes).

- Python 2.6, <http://www.python.org> (you will probably have this already) - Note: PyARTS does not work with Python 3.
- A fortran compiler
- NumPy 1.3.0

- SciPY 0.7.2
- matplotlib 0.99.3 <http://matplotlib.sourceforge.net/>
- To build the documentation: docutils (Ubuntu: python-docutils)

Installation

Once you have all of the above prerequisites installed, and checked out PyARTS from the svn repository, run the following from the base directory.

python setup.py install --user

This will install the package in ~/.local/lib and also it will put some required scripts in ~/.local/bin. Do not try to add the PyARTS repository to your PYTHONPATH directly. It won't work. You need to install PyARTS first, because some parts need to be compiled (such as the *tmatrix* module). If you omit the --user argument python will try and install the modules in the standard 3rd party location (something like /usr/lib/python2.6/site-packages), which obviously won't happen unless you have superuser privileges

In most cases the install command above will work, however, if it cannot find numpy or scipy, you will need to tell it where to find it:

python setup.py build_src build_ext --include-dirs=<wherever>/include/python install --user

Once installed you should make sure that the PATH environment variable includes ~/.local/bin and the ARTS_PATH environment variable points to the arts executable, if this is not in the PATH already.

Testing your Installation

There are several unit tests in the test/ folder of the distribution. These test both the functionality and the accuracy of the software. To run them all, and check that your installation is OK, type

python testall.py -v

If you would like to contribute to PyARTS, which is definitely encouraged, it is strongly recommended that the above command is run, and that all tests are successful, before committing your changes to svn.

Note: currently (19 May 2011), some tests have been disabled because of significant changes in ARTS since the code was originally written.

Examples

Some example scripts are provided in the examples/ folder. Most of them should work as they only depend on data provided in the data/ folder. Some fail because of changes in ARTS. The testall.py script described above actually verifies that the examples run without error.

Documentation

Most modules in the package have reasonably complete docstring documentation. This means that in an interactive python session, online help on a given PyARTS class or function can be obtained by typing `help(PyARTS_function_or_class)`. Or, in ipython, ?PyARTS_function_or_class.

The docstring documentation can also be viewed in easily navigatable html documents by doing the following:

```
<wherever your python stuff is>/pydoc.py -p 1234
```

and open `http://localhost:1234` in your web browser.

There is a user guide in the doc/ folder of the distribution.

History

PyARTS was originally developed by Cory Davis around 2003--2008. From 2010 and onwards, Gerrit Holl has adapted most of PyARTS to newer versions of ARTS. Some parts were dropped as it was not straightforward to adapt those, but those parts may be reintroduced later.

Acknowledgments

Original acknowledgements by Cory Davis:

> This package was developed mainly while working as a post-doctoral researcher at the University of Edinburgh, funded by the Natural Environment Research Council. The focus of this work was understanding cloud effects on observations made by the Aura Microwave Limb Sounder. The extent of this package was greatly enhanced as part of an ESA funded study: "Development of a Radiative Transfer Model for Frequencies between 200 and 1000 GHz" ESA contract 17632/03/NL/FF. 2003-2005.
>
> Thanks are due to my PI's Robert Harwood and Hugh Pumphrey, and also to Dong Wu and Jonathan Jiang at the Jet Propulsion Laboratory. Thanks are also due to Michael Mishchenko for making the T-matrix code available and to Stephen Warren, Bo-Cai Gao, and Warren Wiscombe for their refractive index code. Lastly many thanks are due to the developers of ARTS.

PyARTS: an ARTS related Python package

Introduction

PyARTS is a python package, which has been developed to complement the Atmospheric Radiative Transfer System - ARTS (http://www.sat.ltu.se/arts/). Although ARTS is very flexible software, it's primary function currently is to perform radiative transfer simulations for a given atmospheric state. PyARTS simplifies the process of creating these atmospheric scenarios, and also provides a front-end to the ARTS software for convenient configuration and execution of ARTS radiative transfer calculations.

PyARTS includes two high-level modules that provide most of the functionality needed for the preparation and execution of ARTS simulations:

clouds produces arbitrarily complex multi-phase multi-habit cloud fields for arts simulations. This includes the generation of single scattering properties of non-spherical ice particles and the generation of particle number density fields for given ice and liquid water content fields. *clouds* also provides convenience functions for producing simple 1D and 3D box cloud scenarios.

The module provides a high-level Cloud class and several lower level objects such as Crystal or Droplet. They can generate both single scattering properties and particle number density fields. As of arts-1-14-139, arts can calculate particle number densities internally and there is less need to do so in PyARTS.

arts contains classes and functions that actually perform ARTS simulations. The ArtsRun class provides general functionality for configuring, performing, and managing the output of ARTS simulations.

There are several lower-level modules that, as well as serving the arts and cloud modules, are also useful in their own right:

arts_file_components Contains an ArtsCodeGenerator class that provides low- and mid-level access to generate ARTS control-files.

arts_geometry Various functions dealing with 3D spherical geometry.

arts_math provides several interpolation, quadrature, and grid creation functions.

arts_scat provides functions and classes for the calculation of single scattering properties of ice and liquid water hydrometeors.

arts_types provides support for the manipulation, loading, and saving in ARTS XML format of some ARTS classes, e.g, ArrayOfGriddedField3, GriddedField3, and also the generation of gaseous absorption lookup tables

artsXML provides general XML input and output that can be used for all ARTS objects.

general a general purpose module that includes simplified pickling/unpickling functions for saving arbitrarily complex python objects, and functions for performing multi-threaded calculations.

physics Some physical constants, Rayleigh Jeans brightness temperature function, and psychrometric functions.

plotting general purpose plotting functions and functions for plotting ARTS related quantities (requires matplotlib)

sli contains the SLIData2 class which generates almost optimal grids for 2D sequential linear interpolation. SLI can be used by the ARTS-MC algorithm for the rapid calculation of incoming radiation at the cloudbox boundary.

io Contains functions to locate and read various kinds of data associated with PyARTS or ARTS, such as the Chevalier data or scattering databases.

An example

Here is a simple example python session that demonstrates what can be done with the PyARTS package. In this case we perform 3D polarized radiative transfer in an atmosphere containing a uniform box shaped cloud.

FIXME: the example below needs updating!

1. First import the most commonly used PyARTS modules.

```
>>> from PyARTS import *
```

2. Start by defining a simple box shaped cloud filled with horizontally aligned oblate spheroids.

```
>>> a_cloud=clouds.boxcloud(ztopkm=14.0,zbottomkm=13.0,lat1=-2.0,lat2=2.0,
... lon1=-2.0,lon2=2.0,cb_size={'np':5,'nlat':5,'nlon':5},
... zfile='PyARTS/data/tropical.z.xml',tfile='PyARTS/data/tropical.t.xml',
... IWC=0.1)
>>> horizontal_plate=clouds.Crystal(ptype=30,NP=-1,aspect_ratio=2.0)
>>> a_cloud.addHydrometeor(horizontal_plate)
<Cloud, hydrometeors=1>
```

3. Generate single scattering data files, and particle number density fields.

```
>>> a_cloud.scat_file_gen(f_grid=[500e9,503e9],num_proc=2)
<Cloud, hydrometeors=1>
>>> a_cloud.pnd_field_gen('pnd_field.xml')
<Cloud, hydrometeors=1>
```

4. Generate grids for ARTS RT simulation. For the pressure grid, latitude grid and longitude grid, a fine grid spanning the cloudbox is merged with a course grid covering the modelled atmosphere.

```
>>> p_grid=arts_math.gridmerge(arts_math.nlogspace(101325.0,0.1,100),
... a_cloud.p_grid[1:-2])
>>> artsXML.save(p_grid,'p_grid.xml')
>>> lat_grid=arts_math.gridmerge(arts_math.nlinspace(-16.0,16.0,100),
... a_cloud.lat_grid[1:-2])
>>> artsXML.save(lat_grid,'lat_grid.xml')
>>> lon_grid=lat_grid
>>> artsXML.save(lon_grid,'lon_grid.xml')
```

5. Now define parameters for ARTS run, giving the Monte Carlo algorithm a maximum execution time of 10 seconds (you can also specify a desired accuracy or a fixed number of iterations)

```
>>> arts_params={
...     "atm_basename":"PyARTS/data/tropical",
...     "cloud_box":a_cloud.cloudbox,
...     "freq":501.18e9,
...     "abs_species":["ClO","O3","H2O,H2O-MPM89","N2-SelfContStandardType"],
...     "abs_lookup":"PyARTS/data/gas_abs_lookup.xml",
...     "lat_grid":"lat_grid.xml",
...     "lon_grid":"lon_grid.xml",
...     "max_time":10,
...     "p_grid":"p_grid.xml",
...     "pnd_field_raw":a_cloud.pnd_file,
...     "rte_pos":{'r_or_z':95000.1,'lat':9.1,'lon':0},
```

```
...      "rte_los":{'za':99.14,'aa':180},
...      "scat_data_file":a_cloud.scat_files,
...      "mc_seed": 42,
...      "stokes_dim":4
...      }
```

6. And perform RT calculations (using 2 processors)...

```
>>> my_run=arts.ArtsRun(arts_params,'mcgeneral','cfile.arts')
>>> my_run.run_parallel(2)
<ArtsRun cfile.arts>
```

7. And here is the simulated Stokes vector...

```
>>> print my_run.output['y']
[  1.17613500e+02   5.57757000e+00  -7.19482500e-02  -2.69899500e-01]
>>> print my_run.output['mc_error']
[ 1.66664512  1.23597316  0.4771547   0.43988175]
```

The arts module

PyARTS is designed as a wrapper for ARTS - ie an alternative to writing a control file for each arts run. The module assumes that arts is in the default path. This can be overridden by setting the environment variable ARTS_PATH (e.g. /home/user/test/myarts, where myarts is the name of the executable).

The most useful class is ArtsRun.

ArtsRun objects

Class

A class representing a single arts simulation.

Initialisation

params: mapping Full parameters. For documentation, see arts_file_components.ArtsCodeGenerator.

run_type: string-like Determines the type of run. ArtsRun will initiate an ArtsCodeGenerator by calling the classmethod constructor ArtsCodeGenerator.do_run_type. For example, if run_type is doit_batch, ArtsRun will initiate ArtsCodeGenerator.do_doit_batch. Any do_x method in arts_file_components.ArtsCodeGenerator, e.g. clear, montecarlo, doit_batch, will work here. Different

filename: string (optional) The name of the generated arts control file. If not given, a random, temporary file is used.

Some examples for using this class are given in the docstring for the PyARTS package (__init__.py)

Attributes

params: mapping Stores parameters passed to constructor.

run_type: string Stores run_type passed to constructor.

filename: string Stores the filename used .

code: ArtsCodeGenerator object Stores the instance of ArtsCodeGenerator, after executing .file_gen().

proc: subprocess.Popen object Stores the pipe to the arts process, after running .start().

output: dictionary Dictionary storing all variables output by ARTS. Available after running .process_output().

time_elapsed: float Stores run-time (user space) in seconds.

Selected methods

run arts.ArtsRun.run(self) Run ARTS according to the settings when instantiating ArtsRun.

This method calls self.start() and self.process_output(). The total runtime is stored in self.time_elapsed. Returns self.

start arts.ArtsRun.start(self) Start the arts run.

Sets self.proc to the subprocess.Popen object to handle output etc.

process_output arts.ArtsRun.process_output(self) Wait for ARTS to complete and process output (stdout, stderr)

For all variables output by the controlfile (available in self.code.out), retrieve the value and put them in a hash self.output.

Other module contents

Selected Functions

pnd_fieldCalc arts.pnd_fieldCalc(pnd_field_raw_file, cloudbox, p_file, lat_file, lon_file, pnd_field_file, cloudbox_limits_file=”) Use ARTS to calculate pnd_field.

Uses arts to calculate the pnd_field WSV, which is interpolated onto the arts atmospheric grids.

ppathCalc arts.ppathCalc(argdict) Use ARTS to calculate 3D propagation path.

WARNING: Currently not working!

Uses arts to calculate a 3D propagation path, with atmospheric field settings as described in argdict. The returned object is a dictionary holding all the Ppath member data.

scat_data_monoCalc arts.scat_data_monoCalc(scat_data_raw_file, freq, scat_data_mono_file) Use ARTS to calculate *scat_data_mono*.

Uses arts to calculate the scat_data_mono WSV, which is interpolated at frequency freq.

xml_ascii_to_binary arts.xml_ascii_to_binary(old_file, var_name, new_file) Use ARTS to convert from xml to binary.

xml_binary_to_ascii arts.xml_binary_to_ascii(old_file, var_name, new_file) Use ARTS to convert from binary to xml.

create_incoming_lookup arts.create_incoming_lookup(arts_params, zkm0, za0) Initialises an SLIData2 object suitable for the generation of the ARTS WSV mc_incoming. See examples/mc_incoming_gen.py

IWP_cloud_opt_pathCalc arts.IWP_cloud_opt_pathCalc(settings) Calculates FOV averaged ice water path and FOV averaged optical path. The standard error for each calculation (Monte Carlo for Gaussian antennas) Returns iwp,tau,iwp_error,tau_error

The clouds module

Clouds

Represent 3D clouds and cloud microphysics.

This module includes functions and classes representing 3D clouds and cloud microphysics. This module has all you need for the generation of scattering data files (via the arts_scat module) and particle number density fields, which enable the representation of 3D cloud fields in ARTS simulations

Example of use: a 3D ice and liquid cloud field.

1. load iwc and lwc fields from TRMM data

```
>>> from PyARTS import *
>>> iwc_field=arts_types.LatLonGriddedField3.load('../016/iwc_field.xml')
>>> lwc_field=arts_types.LatLonGriddedField3.load('../016/lwc_field.xml')
```

2. load temperature field I prepared earlier

```
>>> t_field=arts_types.LatLonGriddedField3.load('t_field.xml')
```

3. Define hydrometeors

```
>>> ice_column=clouds.Crystal(NP=-2,aspect_ratio=0.5,ptype=30,npoints=10)
>>> water_droplet=clouds.Droplet(c1=6,c2=1,rc=20)
```

4. Create cloud field

```
>>> a_cloud=clouds.Cloud(t_field=t_field,iwc_field=iwc_field,lwc_field=lwc_field)
```

5. add hydrometeors

```
>>> a_cloud.addHydrometeor(ice_column,habit_fraction=1.0)
>>> a_cloud.addHydrometeor(water_droplet,habit_fraction=1.0)
```

6. generate (or find existing) single scattering data

```
>>> a_cloud.scat_file_gen(f_grid=[200e9,201e9],num_proc=2)
```

7. generate pnd fields

```
>>> a_cloud.pnd_field_gen('pnd_field.xml')
```

8. save cloud object for later

```
>>> quickpickle(a_cloud,'Cloud3D.pickle')
```

Cloud objects

Cloud class

High-level class representing a cloud.

A high level class for the generation of ARTS cloud field data. A Cloud object is initialised with up to three arts_type.LatLonGriddedField3 objects representing 3D temperature, ice water content, and liquid water content fields. The temperature field is compulsory but either IWC or LWC may be omitted. Droplet or Crystal objects can then be added to the Cloud Object using the addHydrometeor method. The user is encouraged to create their own Hydrometeor classes (all that is required is that they have scat_calc and pnd_calc methods with the same input/output arguments). The scat_file_gen and pnd_field_gen methods create the single scattering data files and particle number density files required to represent the cloud field in ARTS simulations.

Note that, as of ARTS-1-14-139, ARTS can internally calculate particle number density fields. The single-scattering-data calculations, however, are still required to do externally.

Constructor input

t_field: LatLonGriddedField3 Temperature field [K]

iwc_field: LatLonGriddedField3 Ice water content field [g/m^3] Maybe omitted, but then lwc_field must be present.

lwc_field: LatLonGriddedField3 Liquid water content field [g/m^3] Maybe omitted, but then iwc_field must be present.

Attributes Note: not all attributes are documented here, only those that are of relevance for the user.

t_field: LatLonGriddedField3 As passed on to the constructor

iwc_field: LatLonGriddedField3 As passed on to the constructor

lwc_field: LatLonGriddedfield3 As passed on to the constructor

p_grid: 1-D array Pressure grid obtained from iwc_field if present, otherwise lwc_field [Pa].

lat_grid: 1-D array Latitude grid, obtained as for p_grid [degrees]

lon_grid: 1-D array Longitude grid, obtained as for lon_grid [degrees]

scat_files: list of strings List of scattering data files generated; filled by .scat_file_gen().

pnd_fields: list of string List of pnd field data files generated; filled by .pnd_field_gen().

pnd_data: ArrayOfLatLonGriddedField3 Contains all the pnd data, as calculated by .pnd_field_gen()

Selected methods

addHydrometeor clouds.Cloud.addHydrometeor(self, hydrometeor, habit_fraction=1.0) Adds hydrometeor to cloud.

Adds a hydrometeor (e.g. a Droplet or Crystal object) to the cloud object. The habit_fraction argument allows the implementation of multi-habit ice clouds. The habit_fractions for all of the added Crystal objects should add up to 1.0. Otherwise the specified iwc_field will not be reproduced.

Parameters

hydrometeor : Anything that has .scat_calc and .pnd_calc, like the MonoCrystal, Crystal and Droplet objects do.

habit_fraction (float between 0 and 1)

Fraction of total cloud that this particle type has.

pnd_field_gen clouds.Cloud.pnd_field_gen(self, filename) Calculates pnd_field_raw and stores it to "filename".

Calculates the pnd data required to represent the cloud field in an ARTS simulation. This produces an arts_types.ArrayOfLatLonGriddedField3 object, which has the same number of elements as the scat_files data member. This is stored in the pnd_data member and output to *filename* in ARTS XML format.

Parameters

filename: stream or string Either a stream (something with a .write method) or a string describing a filename to write the data to.

scat_file_gen

clouds.Cloud.scat_file_gen(self, f_grid, za_grid=array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110
130, 140, 150, 160, 170, 180]), aa_grid=array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180]), num_proc=1)

> **system-message**
> WARNING/2 in `userguide.rst`, line 794
> Definition list ends without a blank line; unexpected unindent. backrefs:

Calculates single scattering data and generate files.

Calculates all of the single scattering data files required to represent the cloud field in an ARTS simulation. The file names are stored in the scat_files data member. The input arguments are f_grid,T_grid,za_grid, and aa_grid: numpy arrays determining the corresponding data in the arts_types.SingleScatteringData objects. The optional argument num_proc determines the number of processes used to complete the task.

Parameters

f_grid (**1D-array**)

Frequency grid for scattering calculation [Hz].

za_grid: 1D-array, optional Zenith angle grid for scattering calculation [degree]. Defaults to every 10 degrees.

aa_grid: 1D-array, optional Azimuth angle grid for scattering calculation [degree]. Defaults to every 10 degrees.

num_proc: int, optional Number of processors to use in calculation. Defaults to 1.

Hydrometeor objects

`Crystal`

Produces scattering data and pnd fields for ice clouds.

The size distribution is the McFarquhar- Heymsfield 1997 distribution (see clouds.mh97, as used in the EOSMLS cloudy-sky forward model. A Crystal object is initialised with the particle parameters ptype, aspect_ratio, NP, which have the same meaning as in arts_types.SingleScatteringData objects. equivalent particle radii and pnd values are determined from the abscissas and weights for an *npoints* Laguerre Gauss quadrature, to give *npoints* arts_types.SingleScatteringData objects, and a pnd field corresponding to these particles. The methods scat_calc and pnd_calc are called by the parent Cloud object.

Parameters

ptype: integer As for arts_types.SingleScatteringData

aspect_ratio: float As for arts_types.SingleScatteringData

NP: integer As for arts_types.SingleScatteringData

npoints: ? (FIXME GH 2011-03-25: what is this?)

T_grid: 1D-array As for arts_types.SingleScatteringData

`Droplet`

Represents a floating particle consisting of liquid water.

Produces scattering data and pnd fields for liquid water clouds. The size distribution is the modified gamma distribution of Nioku, as used in the EOSMLS cloudy-sky forward model. A Droplet object is initialised with the distribution parameters c1, c2, and rc. Some suggested values for these parameters are:

- Stratus (rc=10 micron, c1=6, c2=1)
- Cumulus Congestus (rc=20 micron, c1=5, c2= 0.5).

Scattering properties are integrated over the size distribution using an *npoints* Laguerre Gauss quadrature, to give a one `arts_types.SingleScatteringData object`. *T_grid* is an optional initialisation argument which overides the default temperature grid ([260,280,300,320]) for scattering data file generation. The pnd field is then simply scaled by the lwc_field. The methods scat_calc and pnd_calc are called by the parent Cloud object.

Parameters

c1: number Nioku distribution parameter (?)

c2: number Nioku distribution parameter (?)

rc: number Particle radius [micrometer]

npoints: number ?

T_grid: 1-D array temperature grid for scattering calculations [K]

`Gamma`

Scattering data and pnd fields for ice or liquid.

Produces scattering data and pnd fields for a gamma distribution of either ice or liquid water clouds. The size distribution function is given by $n(r) = N_0 \frac{(\frac{r}{\beta})^{\gamma-1}\exp(-\frac{r}{\beta})}{\beta\Gamma(\gamma)}$,where γ is the shape parameter, and $\beta = \frac{r_{eff}}{\gamma+2}$

A Gamma object is initialised with the distribution parameters r_eff (effective radius), and shape parameter g(γ). *T_grid* is required, as is the phase argument ('ice','liquid'). These two arguments need to be consistent. Scattering properties are integrated over the size distribution using an *npoints* Laguerre Gauss quadrature, to give a one arts_types.SingleScatteringData object. The pnd field is then simply scaled by the iwc_field (or) lwc_field. The methods scat_calc and pnd_calc are called by the parent Cloud object.

`MonoCrystal`

A monodisperse ice particle.

Produces scattering data and pnd fields for ice clouds with particles of only one size.

Parameters

ptype: integer As for arts_types.SingleScatteringData

aspect_ratio: float As for arts_types.SingleScatteringData

NP: integer As for arts_types.SingleScatteringData

equiv_radius: float As for arts_types.SingleScatteringData

T_grid: 1D-array As for arts_types.SingleScatteringData

Other module contents

Selected functions

boxcloud clouds.boxcloud(ztopkm, zbottomkm, lat1, lat2, lon1, lon2, cb_size, zfile, tfile, IWC=None, LWC=None) Return a box shaped Cloud object. ztopkm,zbottomkm,lat1,lat2,lon1,lon2, IWC, LWC are Numeric values, cb_size is a dictionary with keys 'np','nlat', and 'nlon'

gamma_dist clouds.gamma_dist(r, CWC, r_eff, g, phase='ice') Gamma distribution. Returns an array of particle number densities, for an array of particle radii, r, according to the Gamma size distribution,

$$n(r) = N_0 \frac{(\frac{r}{\beta})^{\gamma-1}\exp(-\frac{r}{\beta})}{\beta\Gamma(\gamma)}$$

,where γ is the shape parameter, and $\beta = \frac{r_{eff}}{\gamma+2}$.

Arguments

r: 1-D array array of particle radii [micrometer]

IWC: number cloud water content [gm^{-3}]

r_eff: number effective radius [?]

g: number shape parameter (γ)

mh97 clouds.mh97(IWC, r, TK) MH97 McFarquhar and Heymsfield 1997's particle size distribution

Usage

n,integrated_n=mh97(IWC,r,TK).

Arguments

IWC: number Ice water content [gm^{-3}]

r: 1-D array Array of radii [micrometer]

TK: number Temperature [K]

n: number number density in l^{-1}mm^{-1} (or m$^{-3}\mu$m^{-1});

integrated_n: integrated number density in each size bin in m^{-3}.

Equation

The MH97 size distribution is given by the sum of a gamma distribution for small particles and a log-normal distribution for large particles. The small particle gamma component is

$$N(D_m) = \frac{6IWC_{<100}\alpha^5_{<100}D_m}{\pi\rho_{ice}\Gamma(5)}\exp\left(-\alpha_{<100}D_m\right)$$

,

and the large particle lognormal component is

$$N(D_m) = \frac{6IWC_{>100}}{\pi^{1.5}\rho_{ice}\sqrt{2}\exp\left(3\mu{>}\,100 + 4.5\sigma^2_{>100}\right)D_m\sigma_{>100}D_0^3}\exp\left[-\frac{1}{2}\left(\frac{\log\frac{D_m}{D_0} - \mu{>}\,100}{\sigma_{>100}}\right)^2\right]$$

For details see [McFarquharHeymsfield97].

nioku clouds.nioku(LWC, r, rc, c1, c2) The modified gamma size distribution for water droplets as given in the MLS cloudy sky ATBD. The units are l-1mm-1 (or $m^{-3}\mu m^{-1}$)

$$n(r) = Ar^{c_1} \exp\left(-Br^{c_2}\right)$$

, where

$$A = \frac{3LWCc_2 B^{\frac{c1+4.0}{c2}} \times 10^{12}}{4\pi\Gamma\left(\frac{c1+4.0}{c2}\right)}$$

,and $B = \frac{c_1}{c_2 r_c^{c2}}$

Arguments:

- ***LWC*,** ice water content in gm-3;
- ***r*,** Array of radii in microns;
- ***rc,c1,c2*,** size distribution paramaters. Some suggested values for these parameters are: Stratus (rc=10 micron, c1=6, c2=1), Cumulus Congestus (rc=20 micron, c1=5, c2= 0.5).

Algorithm Description and Theoretical Basis

How Cloud and Hydrometeor objects interact

The Cloud class, and the various hydrometeor classes (see Hydrometeor objects), aim to make it easy to construct arts scenarios from IWC/LWC fields, such as those obtained from NWP/GCM model output, and to be flexible and simple in the application of different microphysical regimes.

Examination of the Cloud source code reveals a very simple class. A Cloud object contains a data member *hydrometeors* which is a list of hydrometeor objects. The Cloud.scat_file_gen and Cloud.pnd_field_gen methods simply iterate through this list, calling the *scat_calc*, and *pnd_calc* methods of each hydrometeor object. This results in a list of scattering data files, and particle number density field for the scenario. The only requirement for the hydrometeor objects is that they have the methods *scat_calc*, and *pnd_calc* methods, which have the same arguments as e.g. Crystal objects...

scat_calc

clouds.Crystal.scat_calc(self, f_grid, za_grid=array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 1
130, 140, 150, 160, 170, 180]), aa_grid=array([0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180]), num_proc=1)

> **system-message**
> WARNING/2 in `userguide.rst`, line 1097
> Definition list ends without a blank line; unexpected unindent. backrefs:

Calculates single scattering data.

Produces npoints scattering data objects, and returns a list (length=npoints) of scattering data files.

Parameters

f_grid: 1D-array frequency grid for scattering calculations [Hz?]

za_grid: 1D-array zenith angle grid [degree]

aa_grid: 1D-array azimuth angle grid [degree]

num_proc: int Number of processors to use in the calculation

Returns

scat_files: list of strings Each string contains the path to a arts-xml-file. The contents of those files describe the single-scattering-properties. The number of strings is determined by npoints (as passed to the constructor).

pnd_calc clouds.Crystal.pnd_calc(self, LWC_field, IWC_field, T_field) Calculates particle number densities.

Returns a list (length=npoints) of pnd fields (GriddedField3), given LWC, IWC and temperature fields (all GriddedField3)

Parameters

LWC_field: LatLonGriddedField3 Liquid water content field [g/m^3]

IWC_field: LatLonGriddedField3 Ice water content field [g/m^3]

T_field: LatLonGriddedField3 Temperature field [K]

Returns

pnd_list list of LatLonGriddedField3 containing particle number density data [g/m^3]. The number of pnd-fields is defined by npoints (see class documentation).

This system should allow for the straightforward implementation of new user defined micro-physical regimes.

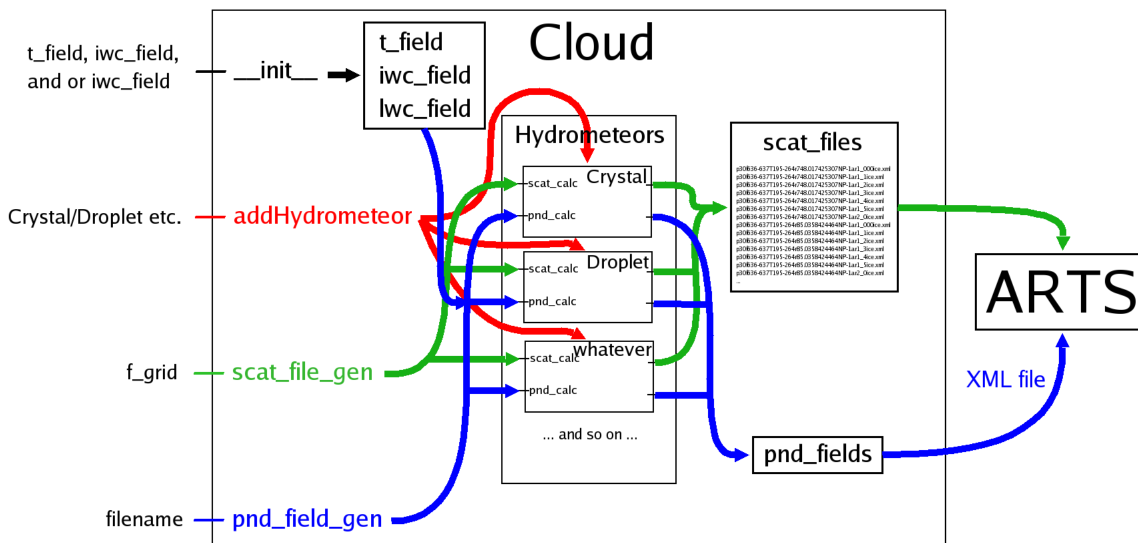

Figure 1: The Cloud class.

Using Laguerre - Gauss Quadrature to represent scattering properties of particle polydispersions

Previously the method for calculating particle number densities (PND) has been sub-optimal. We arbitrarily chose a set of particle sizes, and took bin boundaries between them to give our size bins. The particle size distribution function was then integrated over the size bin to give the particle number density for each size. These PNDs were then all scaled so that IWC was conserved. This method is inelegant : there is no satisfactory way of determining the sizebins / bin points, which led to the choice of a large number (40) of size bins for safety, and is unnecessarily costly.

The theory of Gaussian quadrature states that for an N point method, the approximation,

$$\int_0^\infty x^a \exp(-x) f(x) dx \cong \sum_{i=1}^{N} w_i f(x_i) \tag{1}$$

, is *exact* if $f(x)$ is a polynomial of order up to $2N-1$. The weighting function on the left is closely related to Gaussian distribution and modified Gaussian distributons often found in cloud particle size distributions. The x^a term can accomodate some of the radial dependency (eg r^2, r^3, r^6) of single scattering properties.

Given that our particle number densities are used to calculate some single scattering property Φ, for a polydisperion with some size distribution function $n(r)$, then in ARTS we will be calculating

$$\begin{aligned} \int_0^\infty n(r)\Phi(r)dr &= \int_0^\infty \frac{n(r)}{x^a \exp(-x)} x^a \exp(-x) \Phi(r) \frac{dr}{dx} dx \\ &\cong \sum_{i=1}^{N} \frac{w_i n(r_i)}{x_i^a \exp(-x_i)} \left(\frac{dr}{dx}\right)_i \Phi(r_i) \end{aligned} \tag{2}$$

, where r and x are related by a simple transformation, the exact form of which is determined by the size distribution. The method will be most successful if $\frac{n(r)}{x^a \exp(-x)} \frac{dr}{dx} \Phi(r)$ can be well approximated by a polynomial. Eq. 2 suggests that we represent the polydispersion using a set of N particles with sizes given by the Gauss-Laguerre abscissa, x_i, and for each particle, i, the particle number density is given by

$$PND_i = \frac{w_i n(r_i)}{x_i^a \exp(-x_i)} \left(\frac{dr}{dx}\right)_i \tag{3}$$

Calculation of abscissas and weights for Gauss-Laguerre quadrature is done using the `scipy` function `special.laguerre`

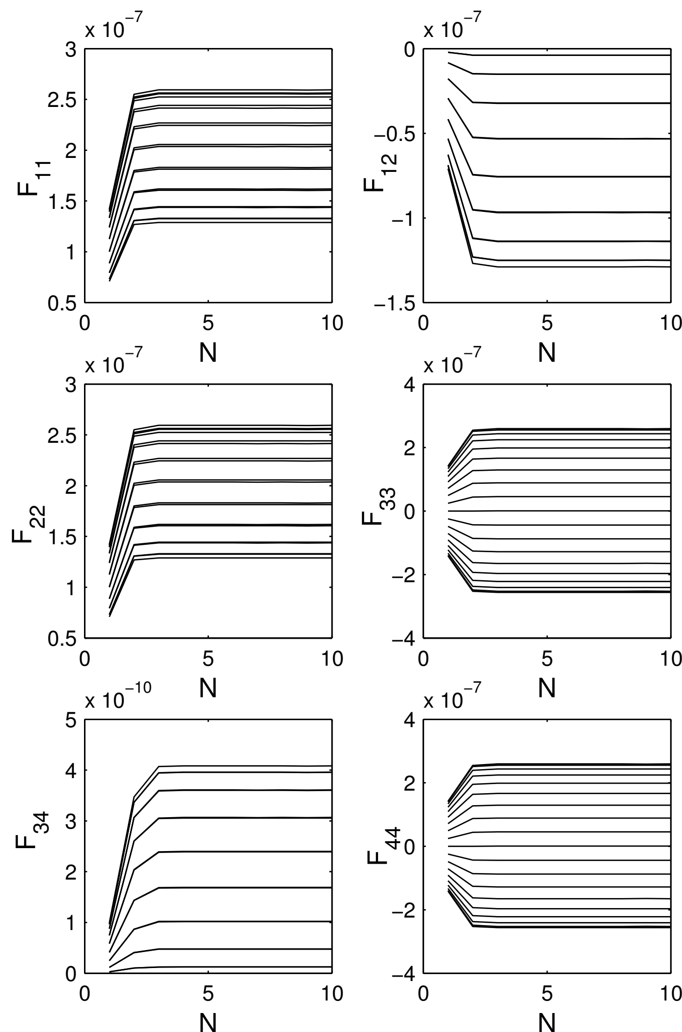

Figure 2: The scattering matrix for liquid spheres, with a liquid water content of 1 gm^{-3}, using a modified gamma size distribution. The x- axes represent the number of quadrature points, and the different lines on each plot are for different scattering angles.

A demonstration Figure 1 indicates that 3 quadrature points (and hence particle types in ARTS) is sufficient for calculating the single scattering properties of liquid water clouds obeying a modified gamma distribution. Reducing the number of particles needed in ARTS simulations improves performance of both ARTS-MC and ARTS-DOIT scattering modules.

Implementation in `Droplet` and `Gamma` objects `Gamma` hydrometeors, and `Droplet` hydrometeors, which use a modified gamma size distribution, are economical because the non-linear factors in the size distribution function are considered independent of the atmospheric field variables (IWC, LWC, and T). This means clouds have the same normalised size distribution at all positions, where size distribution is then scaled to give the correct LWC or IWC. This allows us to use a single *particle type*, with the scattering properties corresponding to an IWC/LWC of 1 gm^{-3}. The PND field is then identical to the IWC/LWC field.

Implementation in `Crystal` objects `Crystal` hydrometeor objects use the MacFarquhar and Heymsfield (1997) size distribution which was obtained from aircraft measurements in tropical cirrus. This correlation (see clouds.mh97) is clearly more complicated than the exponential form best suited for Laguerre Gauss quadrature. However Laguerre Gauss quadrature still seems a good choice given MH97's use of the gamma distribution for small particles. For ARTS we require a

finite, and as small as possible, number of particle types. In Eq. 2 we use the transformation $x = 2\alpha r$. Since the exponential term in the MH97 gamma component depends on IWC we have to choose a suitable value for α, that will give accurate quadrature for the range of IWC encountered, using a minimum number of quadrature points (particle types). The likely range of values for $\alpha_{<100}$ in MH97, led to the choice of $\alpha = 0.02$. A simple test, involving calculating IWC using Laguerre Gauss quadrature for a range of input IWC, showed that $\alpha = 0.02$ resulted in errors of 1% for IWC=1 gm^{-3} and N=4, and 1% for IWC=0.1 gm^{-3} and N=7. By default the Crystal class uses N=10 quadrature points (particle types).

The arts_scat module

ARTS Single Scattering Data calculations.

Contains low-level function to calculate single-scattering properties. The higher-level classes are arts_types.SingleScatteringData and the various types in the clouds module.

SingleScatteringData objects

Class

The class representing the arts SingleScatteringData class.

The data members of this object are identical to the class of the same name in ARTS; it includes all the single scattering properties required for polarized radiative transfer calculations: the extinction matrix, the phase matrix, and the absorption coefficient vector. The angular, frequency, and temperature grids for which these are defined are also included. Another data member - *ptype*, describes the orientational symmetry of the particle ensemble, which determines the format of the single scattering properties. The data structure of the ARTS SingleScatteringData class is described in the ARTS User Guide.

The methods in the SingleScatteringData class enable the calculation of the single scattering properties, and the output of the SingleScatteringData structure in the ARTS XML format (see example file). The low-level calculations are performed in arts_scat.

Constructor input

ptype (**integer**)
: See the User Guide (if you are reading this through the User Guide, see below)

f_grid (**1-D array**)
: f_grid array for frequency grid [Hz]

T_grid (**1-D array**)
: T_grid array for temperature grid [K]

za_grid (**1-D array**)
: za_grid array for zenith-angle grid [degree]

aa_grid (**1-D array**)
: aa_grid array for azimuth-angle grid [degree]

equiv_radius (**number**)
: equivalent volume radius [micrometer]

NP (**integer or None**)
: code for shape: -1 for spheroid, -2 for cylinder, positive for chebyshev, None for arbitrary shape (will not use tmatrix for calculations)

phase (**string**)
: ice, liquid

aspect_ratio (**number**)
: Aspect ratio [no unit]

Some inputs have default values, see SingleScatteringData.defaults.

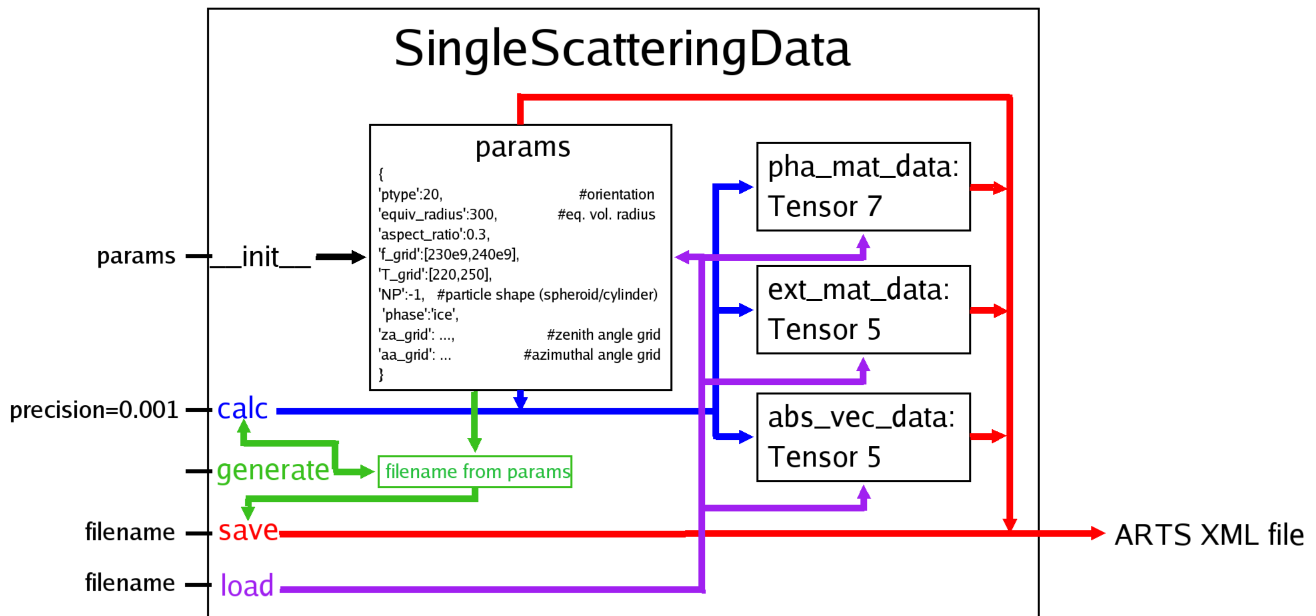

Figure 3: The SingleScatteringData class.

Selected methods

__init__ arts_types.SingleScatteringData.__init__(self, params={}, **kwargs) A SingleScatteringData object is initialised with keyword arguments. See the class documentation for details.

> **system-message**
> WARNING/2 in `userguide.rst`, line 1375
> Inline strong start-string without end-string.

calc arts_types.SingleScatteringData.calc(self, precision=0.001) Calculates the extinction matrix, phase matrix, and absorption vector data required for an arts single scattering data file.

generate arts_types.SingleScatteringData.generate(self, precision=0.001) performs *calc()* and *save* with a filename generated from particle parameters

load arts_types.SingleScatteringData.load(cls, filename) Loads a SingleScatteringData object from an existing file.

Note that this can only import data members that are actually in the file - so the scattering properties may not be consistent with the *params* data member.

save arts_types.SingleScatteringData.save(self, f, compressed=None) Writes data to file

Other module contents

Selected functions

combine arts_scat.combine(scat_data_list, pnd_vec) Returns a single SingleScatteringData object obtained by summing over a list of SingleScatteringData objects, each one multiplied by the corresponding element in a vector of particle number densities. Currently this function requires that all members of scat_data_list have the same value of ptype, and the same angular grids

refice arts_scat.refice(freq, temp) A wrapper for the REFICE fortran program - this time with frequency and temperature as the arguments, and a python exception is raised if inputs are out of range. freq in Hz, temp in K

refliquid arts_scat.refliquid(freq, temp) Calculates the refractive index of liquid water, according to a model based on those of Liebe and Hufford, as used in the EOSMLS scattering code. This has been checked with Table C-1 in the cloudy ATBD. freq in Hz, temp in K

tmat_fxd arts_scat.tmat_fxd(equiv_radius, aspect_ratio, NP, lam, mrr, mri, precision, use_quad=0) A simplified interface to the tmatrix.tmatrix function

tmat_rnd arts_scat.tmat_rnd(equiv_radius, aspect_ratio, NP, lam, mrr, mri, precision, nza, use_quad=0) A simplified interface to the tmd.tmd function

phasmat arts_scat.phasmat(LAM, THET0, THET, PHI0, PHI, BETA, alpha) Calculates the phase matrix and returns it in m^2. This requires that the Tmatrix has already been calculated. See arts_scat.extmat for argument descriptions

extmat arts_scat.extmat(NMAX, LAM, THET0, PHI0, BETA, alpha) Calculate the extinction matrix for a given wavelength (LAM), and a propagation direction given by zenith angle (THET0) and azimuthal angle (PHI0) , both in degrees. BETA and alpha give the particles orientation (These angles are defined as in Mishchenkos paper [mishchenko00]). The output is a 1D array with the 7 independent extinction matrix elements [KJJ,K12,K13,K14,K23,K24,K34] in m^2. To call this method you must first call tmat_fxd

Algorithm and Theoretical Basis

Single Scattering Properties

The single scattering data is calculated using the T-matrix method. For details of the T-matrix method, please consult the references mentioned in the following sections.

ptype=20 In the ptype=20 case, where we have completely random orientation, we use a slightly modified version of Mishchenko's random orientation T-matrix code [mishtrav98]. Mischenko's source code has been altered to provide a python extension module, **tmd.so**, which contains the function `tmd`. This function returns the scattering cross-section, C_{sca}, the extinction cross-section, C_{ext}, and the scattering matrix elements, F_{11}, F_{22}, F_{33}, F_{44}, F_{12}, F_{34}. This function is called by the SingleScatteringData.calc() method, and it is not intended for `tmd.tmd` to be called directly by the user. `SingleScatteringData.calc()` calls `tmd.tmd` with arguments suitable for a monodispersion of particle sizes, as this allows a consistent interface for both `ptype`=20 and `ptype=30`. Size distributions of both these particle types can be handled by the clouds module. However, if you want to use the size distribution capabilities of Mishchenko's code, the `tmd.tmd` function can be called directly. See the pydoc documentation for the argument list and [mishtrav98] for the argument definitions.

ptype=30 For ptype=30, where there is horizontal alignment, but random azimuthal orientation, we use a modified version the fixed orientation code of Mishchenko [mishchenko00]. Mischenko's source code has been altered to provide a python extension module, **tmatrix.so**, which contains the functions `tmatrix` and `ampld`. `tmatrix.tmatrix` calculates the T-matrix for given particle and radiation parameters, and stores this in the data structure `tmatrix.tmat`. The function `tmatrix.ampl`, uses the T-matrix data to calculate the scattering amplitude function $\mathbf{S}(\mathbf{n}, \mathbf{n}', \alpha, \beta)$, for given incident, $\mathbf{n}$, and scattered, $\mathbf{n}'$, directions, and orientation angles α, and β. This arrangement makes use of the fact that the T-matrix need only be calculated once for a given particle and frequency, to get single scattering properties for a range of directions and particle orientations. From $\mathbf{S}(\mathbf{n}, \mathbf{n}', \alpha, \beta)$, it is straightforward to get the extinction matrix $\mathbf{K}(\mathbf{n})$, and phase matrix $\mathbf{Z}(\mathbf{n}, \mathbf{n}')$; for details see [mishchenko00]. Again, the `tmatrix` module is not intended for the user; the functions mentioned above are called within `SingleScatteringData.calc()`.

The purpose of the `ptype=30` case in the SingleScatteringData python class is to represent scattering by *horizontally aligned* particles. This means that oblate particles (aspect ratio $>$ 1) have their rotation axis parallel to the local zenith. In the notation of [mishchenko00], this

corresponds to an orientation angle $\beta = 0$, which makes the orientation angle α irrelevant due to the rotational symmetry of the particle. Conversely, prolate particles have the axis of rotation perpendicular to the local zenith. This means that in the case of horizontally aligned prolate particles, scattering properties must be averaged over all possible azimuth orientations, α, with $\beta = \pi/2$.

Orientation Averaging This section only applies to horizontally aligned prolate particles. The orientationally averaged extinction matrix is obtained from the averaged T-matrix, which can be calculated 'exactly' from a single T-matrix calculation according to the analytic method described in [mishchenko91]. This is implemented in the function *tmatrix.avgTmatrix*. Unfortunately the orientationally average T-matrix is **not** useful for calculating the orientationally averaged phase matrix. In short this is because unlike the extinction matrix, phase matrix elements can not be expressed as linear expansions of T-matrix elements. Therefore $\langle \mathbf{Z}(\mathbf{n}, \mathbf{n}') \rangle$ must be obtained by numerical integration.

$$\langle \mathbf{Z}(\mathbf{n}, \mathbf{n}') \rangle = \frac{1}{\pi} \int_0^{\pi} \mathbf{Z}(\mathbf{n}, \mathbf{n}', \beta = \pi/2, \alpha) d\alpha$$

Several quadrature routines have been trialed for this integration. To date, by far the best in terms of accuracy and speed has been Gauss Legendre quadrature [pressetal92]. In this case we use a 10 point quadrature. This is implemented by the gauss_leg function in the arts_math module.

Calculation of the absorption coefficient vector Calculation of the absorption coefficient vector is the most taxing part of the arts_types.SingleScatteringData calculations, particularly for oblate p30 particles. For p20 particles we have simply the absorption cross-section, $K_{a1} = C_{ext} - C_{sca}$, where the values on the RHS are obtained directly from `tmd.tmd`

However, for p30 particles, the absorption coefficient vector is given by

$$\begin{aligned} K_{ai} &= \langle K_{i1}(\mathbf{n}) \rangle - \int_{4\pi} d(\mathbf{n}') \langle Z_{i1}(\mathbf{n}, \mathbf{n}') \rangle \\ &= \langle K_{i1}(\mathbf{n}) \rangle - 2 \int_{\pi} d\Delta\phi \int_{\pi} d\theta' \langle Z_{i1}(\theta, \Delta\phi, \theta') \rangle sin(\theta') \end{aligned} \tag{4}$$

The integration is performed using multi-dimensional Gauss-Legendre quadrature, which is implemented as the multi_gauss_leg function in the arts_math module. In the case of prolate particles, the evaluation of $\langle Z_{i1}(\theta, \Delta\phi, \theta') \rangle$ requires integration over azimuthal orientation. For this reason, the integration in Eq. 4 is done using 6 point Gaussian quadrature, whereas for oblate particles we use the 10 point method.

Refractive Index of Ice and Liquid Water

The calculation of single scattering properties for ice and liquid hydrometeors requires knowledge of the complex refractive index of the material in question. For both ice and liquid water the complex refractive index is a function of temperature and frequency.

PyArts incorporates the fortran code, REFICE.f of Stephen Warren, Warren Wiscombe, and Bo-Cai Gao to calculate the refractive index of ice at a given frequency and temperature. This is most easily accessed by the function `refice` (see above) in the arts_scat module. This function looks up tables based mainly on the tabulated data of [Warrenetal84]. REFICE.f has incorporated data published since Warren's paper, but this is not in the mm-submm range. Stephen Warren has suggested that the data of [MaetzlerWegmueller87], be consulted for the microwave region. This has **not** been implemented in the REFICE extension model.

Figure 2 was generated by the script `plot_refr_ind.py`, which is in the "examples" directory of the PyARTS distribution.

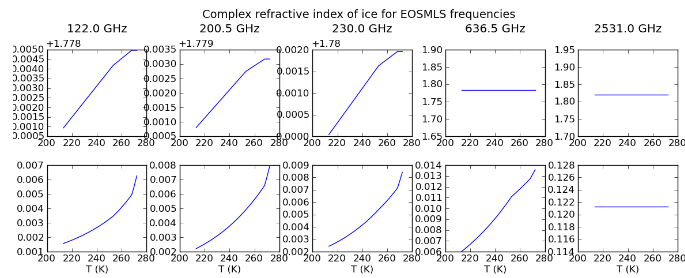

Figure 4: Output of examples/plot_refr_ind.py

For liquid cloud droplets, the complex refractive index is calculated according to the model described in the EOS-MLS Cloudy-Sky ATBD, which is based on the empirical model of [Liebeetal89] and [Hufford91]. This is implemented in the arts_scat.refliquid function described above.

The range of convergent size parameters and aspect ratios for ice crystal optical property generation

The original T-matrix fortran codes have been modified to call subroutines in the LAPACK library. This gives the same extended range of convergent size parameters and aspect ratios as the optional NAG enhancements described in Mishchenko's papers.

Figure 3 shows the minimum integer size parameters, for a range of aspect ratios, that cause convergence failures in the PyARTS implementation of the T-matrix codes. Here the complex refractive index is given by ...

```
>>> from PyARTS.arts_scat import *
>>> T=240 #temperature
>>> f=300e9 #frequency
>>> m=refice(f,T) #refractive index of ice using Warren(1984)
>>> print m
(1.78117084503+0.00504761422053j)
```

The convergence failure parameters shown in Figure 3 were obtained by the script `tmat_limits.py`, which resides in the test folder of the PyARTS dsitribution. For size parameters and aspect ratios on or above the curves in Fig. 3, the T-matrix code will fail to converge.

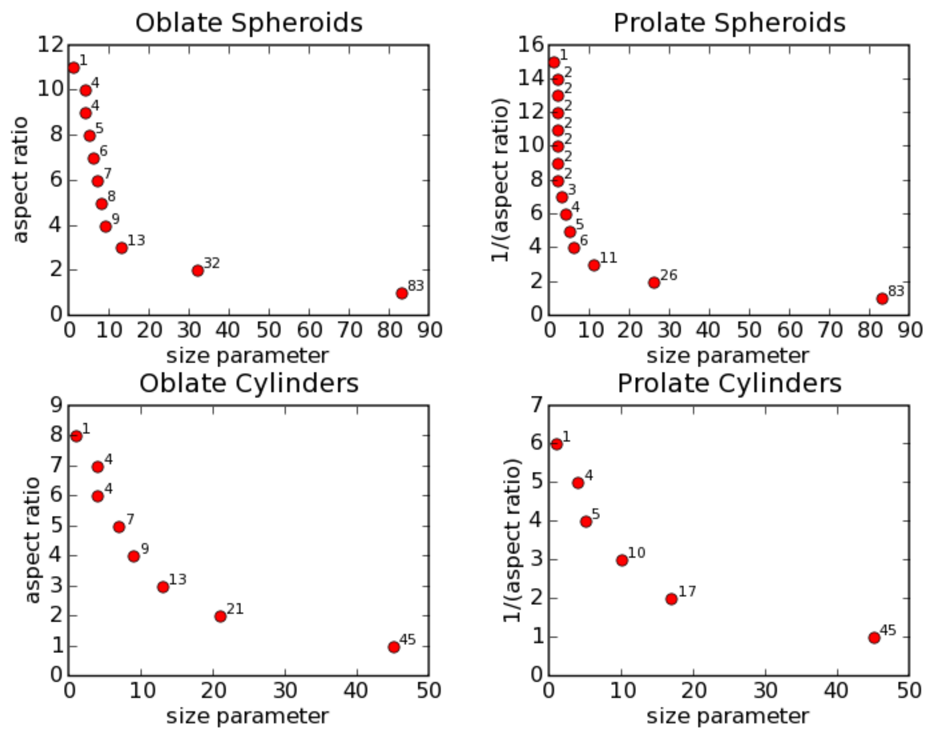

Figure 5: T-matrix convergence failure parameters for m=(1.78117084503+0.00504761422053j)

The arts_types module

The arts_types module includes support for various ARTS-classes.

Due to the dynamic nature of Python, some are implemented in a more generic way. For example, ArrayOf can be easily subclassed to be an array of anything, and all gridded-fields are subclasses of GriddedField.

Classes of special interest may be:

- LatLonGriddedField3: Special case of GriddedField3 for lat/lon/pressure data
- AbsSpecies
- SingleScatteringData
- ScatteringMetaData

This module allows the generation, manipulation, and input/output in ARTS XML format of these objects.

LatLonGriddedField3 objects

A gridded field consists of a pressure grid vector, a latitude vector, a longitude vector and a Tensor3 for the data itself.

Methods

__init__ arts_types.LatLonGriddedField3.__init__(self, p_grid, lat_grid, lon_grid, data) GriddedField3 objects are initialised with a pressure grid vector, a latitude vector, a longitude vector and a Tensor3 for the data itself

__call__ arts_types.LatLonGriddedField3.__call__(self, p_grid, lat_grid, lon_grid) interpolate the field onto new grids, returns only the field data

expandTo3D arts_types.LatLonGriddedField3.expandTo3D(self, new_lat_grid, new_lon_grid) converts the existing 1D field to 3D and returns a new field. The original field is not changed

pad arts_types.LatLonGriddedField3.pad(self, plims=[110000.0, 1.0000000000000001e-05], latlims=[-90, 90], lonlims=[-180, 180]) Adds extra gridpoints at new extremities in all dimensions. data values are copied from the existing end points

load arts_types.LatLonGriddedField3.load(cls, filename) load a GriddedField3 object from an ARTS XML file

save arts_types.LatLonGriddedField3.save(self, filename) Save the GriddedField3 object to an ARTS XML file

The `plotting` module

General purpose plotting functions using the matplotlib package.

`SentinelMap` objects

plotting.SentinelMap has no docstring!

Other module contents

Selected functions

hotcoldmap plotting.hotcoldmap(zmin, zmax) produces a color map with a black-blue-green scale for values below zero, and a black-red-yellow scale for values above zero

myPcolor plotting.myPcolor(x, y, z, **kwargs) With the matplotlib pcolor you actually lose the last row and column of data. This function addresses this, and produces a pcolor plot where the patches are centred on the x and y values

> **system-message**
> WARNING/2 in `userguide.rst`, line 1860
> Inline strong start-string without end-string.

mySubplot plotting.mySubplot(nrows, ncols, pnum, figpos=[0.050000000000000003, 0.050000000000000003, 0.90000000000000002, 0.90000000000000002], axpos=[0.14999999999999999, 0.14999999999999999, 0.75, 0.75]) More like the matlab subplot than matplotlib. Divides a portion of the current figure, determined by *figpos* in to *nrows* by *ncolumns* panels. The normalised position of the axes within the panel is given by *axpos*. The axes object is returned.

drawCloudBox plotting.drawCloudBox(zbase, ztop, lat1, lat2, npts=40, format='k') draws a cloudbox cross section

drawPpath plotting.drawPpath(filename, format='k') plots a propagation path from an ARTS XML file in x,y (km) coordinates

drawSurface plotting.drawSurface(lat1, lat2, npts=40, format='k') draws the geoid surface

setDataAspectRatioByAxisPos plotting.setDataAspectRatioByAxisPos(ax, r) Same idea as matlab. Adjusts the axis position to fix the aspect ratio

setDataAspectRatioByFigSize plotting.setDataAspectRatioByFigSize(ax, r) Same idea as matlab. Adjusts the figure size to fix the aspect ratio

shiftaxes plotting.shiftaxes(ax, delta_pos) For use with matplotlib. **input:**

> **system-message**
> ERROR/3 in `plotting.shiftaxes.rst`, line 7
> Unexpected indentation. backrefs:

ax, a matplotlib.axes object; *delta_pos*, a 4 element list or array correspond to [delta_x_start,delta_y_start,delta
in normalised units.

Demonstration

Figure 4 shows the output of examples/geometry.py, which uses . `plotting.drawCloudBox`, `plotting.drawPpath`, and `plotting.drawSurface`.

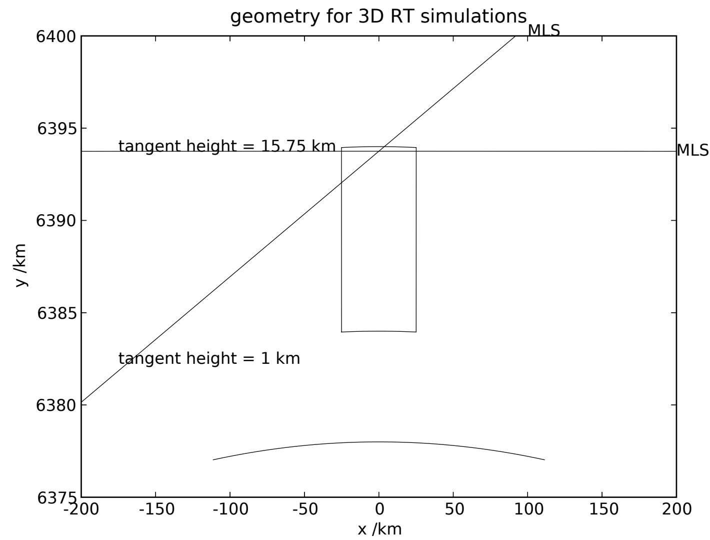

Figure 6: the output of examples/mc_incoming_gengeometry.py, which uses . `plotting.drawCloudBox`, `plotting.drawPpath`, and `plotting.drawSurface`.

The `artsXML` module

artsXML

Creat, load and save datafiles in the Arts XML format.

For XML output, the main class is XMLfile. An XMLfile object is initialised with a filename or a stream.

```
>>> from PyARTS import artsXML
>>> testfile=artsXML.XMLfile('a_test_file.xml')
```

Arts data objects are then added to the file with the add method.

```
>>> a_tensor=ones([3,5,6],numpy.float32)
>>> testfile.add(a_tensor)
```

Context managers are also supported:

```
>>> with artsXML.XMLfile("a_test_file.xml") as tf:
...     tf.add(my_tensor)
```

Then you don't need to worry about closing the file, as the context manager takes care of this.

The ARTS data type to save is is determined automatically from the python type. This mapping is specified by the dictionary artsXML.mapping. For instance, [[1,2,3],[4,5,6]] would be saved as an ArrayOfArrayOfIndex, whereas [array([1,2,3]),array([4,5,6])] would be saved as an ArrayOfVector. Note that it is not guaranteed that ARTS actually understands this! For Tensor type objects, the tag name (eg. Tensor3) and the size attributes are determined automatically by the shape of the numpy array.

The file must then be closed with the close() method:

```
>>> testfile.close()
```

A shortcut save function is available. Using save the above is achieved in one line.

```
>>> artsXML.save(a_tensor,'a_test_file.xml')
```

Some more complicated structures, like SingleScatteringData objects, have their own save methods which utilize this module.

The load function is a low-level function that returns a dictionary structure reflecting the structure of the XML file. If you already know the type of the data you are going to load, you can use the particular constructor, such as SingleScatteringData.load or ArrayOfLatLonGriddedField3.load. All of these will use artsXML.load as a backend.

`XMLfile` objects

arts XML output class. Initialise with a filename or object.

Warning: this will open the file for writing, thus over-writing any existing data!

Selected Methods

artsXML.XMLfile.close(self, really=True) This must be called to finalise the XML file

Other module contents

Selected functions

save artsXML.save(data, filename) Saves data to arts XML file (filename or stream).

If the filename ends in '.gz' the XML file will be gzipped

load artsXML.load(filename, use_names=True) Loads an ArtsXML-file.

This general purpose function returns a dictionary structure reflecting the structure of the XML file. If there is only one object in the structure, then that single object is returned. As far as far as I know, this works with every data type exported by ARTS. Note that only in special simple cases, such as single tensors, will load return exactly the same data given to the save command. The more complicated data structures (e.g. SingleScatteringData) have their own load methods. gzipped files can be loaded (as long as they end in 'gz').

Parameters

filename (**string-like**)

Path to file to load (.xml or .xml.gz)

use_names (**boolean, optional (default: True)**)

Use name tags in XML-file

Returns Returns an OrderedDict with the contents of the XML-file.

The `arts_math` module

This module includes general purpose math functions. This module was developed before scipy was included as a prerequisite, so there will be some functions remaining that duplicate scipy functionality.

Other module contents

Selected functions

gauss_leg arts_math.gauss_leg(func, a, b, n) Gauss legendre integration with n abscissa

multi_gauss_leg arts_math.multi_gauss_leg(func, rangelist, n=10) (no documentation found)

gridmerge arts_math.gridmerge(aa, ba) Merges two sorted vectors(numpy array objects)

locate arts_math.locate(xa, x) Given an array, xa, and a number x, locate returns the index i, such that x lies between xa[i] and xi[j]. Answers of -1 or n-1 indeicate that x is beyond the range of xx

nlogspace arts_math.nlogspace(start, stop, n) Identical to the function of the same name in ARTS; Returns a vector logarithmically spaced vector between start and stop of length n (equals the Matlab function logspace)

The io module

IO

Contains various functions to read particular datasets

Other module contents

Selected functions

read_chevalier io.read_chevalier(tp) Read Chevallier data, both "clear-sky" and cloudy fields. Requires environment variable ARTS_XMLDATA_PATH to be set.

Parameters

type (**string-like**)

What type of Chevalier-data to read. Valid values: 'clear', 'cloud', 'all'.

var (**string-like**)

For what the Chevalier-data is maximised. Valid values: 'ccol', 'oz', 'q', 'rcol', 't'.

Returns nd-array, 5000 x 92, containing "clear-sky" and cloudy fields. Record names as for the Chevalier README:
('p', 'T', 'z', 'VMR_H2O', 'VMR_O3', 'CLW', 'CIW', 'Rain', 'Snow')

readYang2005 io.readYang2005(f) Read Yang-2005 type formatted file and get SingleScatteringData ndarray.

Parameters

f (**string-like**)

Path to Yang-data to be read

Returns Returns (size, wavelength, angles), SSD, e.g. a two-element tuple, the first element having in turn three elements.

size (**1D-array (sizes,)**)

Sizes for the particles [m]

wavelengths (**1D-array (wavelengths,)**)

Wavelengths for the particles [m]

angles (**1D-array (angles,)**)

Zenith angles for the scattering [degrees]

SSD (**2D-array (wavelengths, sizes)**)

An array with shape (wavelengths, sizes), containing the fields:

P11 : first element of the phase matrix [m^2] (angles,)

A_projected : projected area [m]

K : extinction cross section [m^2]

A : absorption cross section [m^2]

density : particle density [kg/m^3]

d_max : maximum diameter [m]

V : volume [m^3]

aspect_ratio : particle aspect ratio [no unit]

See also /storage3/data/scattering_databases/yang/Yang_2005_IR/README

readHong io.readHong(p) Read Hong-type formatted file and get SingleScatteringData ndarray.

Parameters

p (**string-like**)

Path to Yang-data to be read

Returns Returns (size, wavelength, angles), SSD, e.g. a two-element tuple, the first element having in turn three elements.

size (**1D-array (sizes,)**)

Sizes for the particles [m]

wavelengths (**1D-array (wavelengths,)**)

Wavelengths for the particles [m]

angles (**1D-array (angles,)**)

Zenith angles for the scattering [degrees]

SSD (**2D-array (wavelengths, sizes)**)

An array with shape (wavelengths, sizes), containing the fields:

Z (**8 elements of the phase matrix [m^2] (8, angles)**)

P11, P12, P21, P22, P33, P34, P43, P44

K : extinction cross section [m^2]

A : absorption cross section [m^2]

A_projected : projected area [m]

density : particle density [kg/m^3]

d_max : maximum diameter [m]

V : volume [m^3]

aspect_ratio : particle aspect ratio [no unit]

The `general` module

This file includes interpreter or general purpose functions

Selected functions

Other module contents

multi_thread2 general.multi_thread2(func, inarglist, num_proc, logging) executes <func> for every argument list in inarglist and returns a list of output objects. num_proc specifies the desired number of concurrent processes (usually the number of CPUs)

quickpickle general.quickpickle(object, filename) pickle.dump <object> to <filename>. If filename ends with .gz the pickle file is gzipped

quickunpickle general.quickunpickle(filename) pickle.load an object from <filename> If an absolute path is not included in filename, then environment variable DATA_PATH is used. Saves a few lines of code.

The `sli` module

This module defines the SLIData2 class, which allows the creation of optimized grids for 2D sequential linear interpolation, as described by [Changetal97] (pdf).

The main difference with the method used by SLIData2 and that described in the paper, is that we start with a course grid, and every function evaluation is included in the final grid. The motivation for this is that is expected that function evaluations (e.g. ARTS RT simulations) are expensive.

For an example of use, see the arts.create_incoming_lookup function.

`SLIData2` objects

Class for 2D sequential linear interpolation

Selected methods

__init__ sli.SLIData2.__init__(self, func=None, x1=None, x2=None) initialises the SLIData2 object with a standard grid defined by vectors x1 and x2. func must be a function y=func(x1,x2), where x1,x2 and y are vectors of the same length

refine sli.SLIData2.refine(self, N) Refine the grid by increasing the total number of gridpoints to 'about' N. Generally it is good to call refine 2 or more times to successively add more points to the grid.

interp sli.SLIData2.interp(self, x1, x2) interpolate SLIData2 at x1 and x2 (single numeric values only)

plot sli.SLIData2.plot(self) create a simple scatter plot of the grid.

load sli.SLIData2.load(self, filename) reads SLIData2 object from an XML file

save sli.SLIData2.save(self, filename) output the SLIData2 object in ARTS XML format

Demonstration

Figure 5 shows the output of examples/mc_incoming_gen.py, which makes use of the `SLIData2 class` to create an optimised 2D (altitude, zenith angle) lookup table of incoming clear-sky radiance, for MonteCarlo scattering simulations with a pseudo-3D atmosphere (3D cloud, but a 1D atmosphere outside the cloudbox).

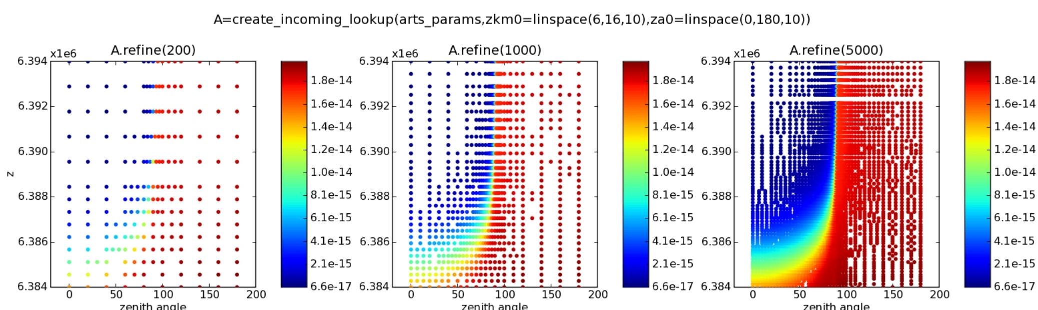

Figure 7: the output of examples/mc_incoming_gen.py, which makes use of the `SLIData2 class`.

References

[Changetal97] Chang, J. Z., J. P. Allebach, C. A. Bouman, Sequential Linear Interpolation of Multidimensional Functions, *IEEE Trans. Image Proc*, 6(9),1997. (pdf)

[Hufford91] Hufford, G., A model for the complex permittivity of ice at frequencies below 1 THz. *Int. J. Infrared Millimeter Waves*, 12, 677-682, 1991.

[Liebeetal89] Liebe, H. J., T. Manabe, and G. A. Hufford, Millimeter-wave attenuation and delay rates due to fog/cloud conditions. *IEEE Trans. Ant. Prop.*, 37, 1617-1623, 1989.

[MaetzlerWegmueller87] Maetzler and Wegmueller, *J. Phys. D.* 20, 1623-1630, 1987

[McFarquharHeymsfield97] G.M. McFarquhar and A.J. Heymsfield, Parametrization of tropical ice crystal size distributions and implications for radiative transfer: Results from CEPEX, *J. Atmos. Sci.*, 54, 2187-2200, 1997.

[mishchenko91] M. I. Mishchenko, Extinction and polarization of transmitted light by partially aligned nonspherical grains, *Astrophysical Journal*, 367, 561-574, 1991. (pdf)

[mishtrav98] Mishchenko, M.I. and Travis, L.D, Capabilities and limitations of a current FORTRAN implementation of the T-matrix method for randomly oriented, rotationally symmetric scatterers., *J. Quant. Spectrosc. Radiat. Transfer*, 60(3), 309-324,1998. (pdf)

[mishchenko00] M.I. Mishchenko, Calculation of the amplitude matrix for a non-spherical particle in a fixed orientation, *Applied Optics*, 39(6), 1026-1031,2000. (pdf)

[pressetal92] H.P Press, S.A Teukolksky, W.T. vetterling, and B.P Flannery, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[Warrenetal84] S.G. Warren, Optical constants of ice from the ultraviolet to the microwave, *Applied Optics*, 23(8), 218-229, 1984.