PyARTS User Guide, Algorithm Description and Theoretical Basis

Cory Davis

date: 2006-09-20

PyARTS version: 1.1.16

email: cdavis@staffmail.ed.ac.uk

Contents

1 Introduction

1.1 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.

1.2 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 (SciPy, matplotlib)
- platform independent
- straightforward package distribution
- **FREE**

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 point, 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.

1.3 Downloading PyARTS

PyARTS is available by anonymous cvs from the ARTS website http://www.sat.uni-bremen.de/arts/.
An up-to-date PyARTS source distribution can also be found at http://www.met.ed.ac.uk/~cdavis/PyARTS/latest

1.4 Prerequisites

- Python 2.3, <http://www.python.org> (you will probably have this already)
- A fortran compiler
- NumPy and SciPy <http://www.scipy.org>
- matplotlib <http://matplotlib.sourceforge.net/>

The versions of the above that I am currently using are numpy-0.9.8 and scipy-0.4.9. Some parts of PyARTS also require matplotlib, where I currently have matplotlib-0.86.2.

1.5 Installation

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

python setup.py install --home=~

This will install the package in ~/lib/python and also it will but some scripts in ~/bin. If you omit the --home argument python will try and install the modules in the standard 3rd party location (something like /usr/lib/python2.2/site-packages), which obviously wont happen unless you have superuser privileges

In most cases the install command above will work, however, if like me, your Numeric package is not installed in the standard place (something like /usr/lib/python2.x/site-packages/Numeric), you need to use a slightly different command to build and install PyARTS:

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

Once installed you should modify your PYTHONPATH environment variable to include the installation directory (eg ~/lib/python).

1.6 Testing your Installation

There are several unit tests in the test/ folder of the distribution. These test both both the functionality and 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 CVS.

1.7 Examples

Some example scripts are provided in the examples/ folder. These all should work as they only depend on data provided in the data/ folder. The testall.py script described above actually verifies that the examples run without error. At the time of writing the examples are:

get_atm_fields.py A demonstration of the artsGetAtmFields function

MCwith3Dboxcloud.py Creates a simple cloudy-sky scenario and performs a single radiative transfer calculation using the ARTS-MC module.

mc_incoming_gen.py demonstrates the use of the create_incoming_lookup function which creates a sequential linear interpolation lookup table of incoming radiances that can be used by the ARTS Monte Carlo radiative transfer algorithm.

geometry.py uses the plotting module to show ARTS RT simulation geometry.

plot_refr_ind.py Uses the arts_scat module, and the plotting module to show the refractive index of ice at several EOS-MLS frequencies

IWP_opt_pathCalc.py uses the arts.IWP_cloud_opt_pathCalc function to calculate the FOV averaged cloud optical path and ice water path for a simple 3D cloud scene

1.8 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)`.

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. A recent version of this document can be found at http://www.met.ed.ac.uk/~cdavis/PyARTS/userguide.pdf

2   PyARTS: an ARTS related Python package

2.1   Introduction

PyARTS is a python package, which has been developed to compliment the Atmospheric Radiative Transfer System - ARTS (http://www.sat.uni-bremen.de/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.

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_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.

2.2   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.

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)
<PyARTS.clouds.Cloud instance at 0x405c8c4c>
```

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

```
>>> a_cloud.scat_file_gen(f_grid=[500e9,503e9],num_proc=2)
<PyARTS.clouds.Cloud instance at 0x405c8c4c>
>>> a_cloud.pnd_field_gen('pnd_field.xml')
<PyARTS.clouds.Cloud instance at 0x405c8c4c>
```

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.saveTensor(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.saveTensor(lat_grid,'lat_grid.xml')
>>> lon_grid=lat_grid
>>> artsXML.saveTensor(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,
...     "gas_species":["ClO","O3","H2O,H2O-MPM89","N2-SelfContStandardType"],
...     "gas_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,
...     "stokes_dim":4
...     }
```

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

```
>>> my_run=arts.ArtsRun(arts_params,'montecarlo','cfile.arts')
>>> my_run.run_parallel(2)
<PyARTS.arts.ArtsRun instance at 0x404585ac>
```

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

```
>>> print 'Simulated Stokes vector =
'+str(my_run.output['y'])
Simulated Stokes vector =
[  1.17613500e+02   5.57757000e+00  -7.19482500e-02  -2.69899500e-01]
>>> print 'standard error = '+str(my_run.output['error'])
standard error = [ 1.66664512  1.23597316  0.4771547   0.43988175]
```

3   The arts module

PyARTS.arts 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. This provides general functionality for performing clear sky and cloudy (ARTS-MC only) ARTS simulations. For an example of the use of ArtsRun, see examples/MCwith3Dboxcloud.py.

3.1   `ArtsRun` objects

A class representing a single arts simulation. The initialisation arguments are

params: a dictionary of parameters. Any "keyword":value pairs not specified in argdict are given default values. default parameters are stored in PyARTS.defaults. Its a good idea to refer to this dictionary for the valid keywords.

run_type: "montecarlo", "mcgeneral", "clear", "clear3D","mcipa","iwp_tau"

filename: the name of the generated arts control file.

Some examples for using this class are given above in the docstring for the PyARTS module

3.1.1   Selected methods

arts.ArtsRun.run (*self*): Simply calls the start() and process_out_stream() methods

arts.ArtsRun.start (*self*): Start the arts run. And initiates a Python file object self.out_stream to handle arts output. Note that arts output is also written to file self.filename + ".out"

arts.ArtsRun.process_out_stream (*self*): Extracts important numerical values from self.std_out

arts.ArtsRun.run_parallel (*self, number_of_processes*): Only for monte carlo simulations. Divides self.params["max_iter"] by number_of_processes and runs number_of_processes arts processes. The results are then combined. This is particularly worthwhile on multiple processor machines.

3.2   Selected Functions

arts.pnd_fieldCalc (*pnd_field_raw_file, cloudbox, p_file, lat_file, lon_file, pnd_field_file, cloudbox_limits_file* =): Uses arts to calculate the pnd_field WSV, which is interpolated onto the arts atmospheric grids

arts.ppathCalc (*argdict*): 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

arts.scat_data_monoCalc (*scat_data_raw_file, freq, scat_data_mono_file*): Uses arts to calculate the scat_data_mono WSV, which is interpolated at frequency freq

arts.xml_ascii_to_binary (*old_file, var_name, new_file*): uses arts to convert an ascii xml file (*old_file*) corresponding to an ARTS workspace variable (*var_name*) to binary

arts.xml_binary_to_ascii (*old_file, var_name, new_file*): uses arts to convert an binary xml file (*old_file*) corresponding to an ARTS workspace variable (*var_name*) to ascii

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

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

4   The `clouds` module

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.GriddedField3().load('../016/iwc_field.xml')
>>> lwc_field=arts_types.GriddedField3().load('../016/lwc_field.xml')
```

2. load temperature field I prepared earlier

```
>>> t_field=arts_types.GriddedField3().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')
```

4.1   `Cloud` objects

A high level class for the generation of ARTS cloud field data. A Cloud object is initialised with up to three arts_type.GriddedField3 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.

4.1.1 Selected methods

clouds.Cloud.addHydrometeor (*self*, *hydrometeor*, *habit_fraction* =1.0): 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.

clouds.Cloud.pnd_field_gen (*self*, *filename* =): Calculates the pnd data required to represent the cloud field in an ARTS simulation. This produces an arts_types.ArrayOfGriddedField3 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.

clouds.Cloud.scat_file_gen (*self*, *f_grid*, *za_grid* =[0 10 20 ...], *aa_grid* =[0 10 20 ...], *num_proc* =1): 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_scat.SingleScatteringData objects. The optional argument num_proc determines the number of processes used to complete the task.

4.2 Hydrometeor objects

4.2.1 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_scat.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_scat.SingleScatteringData objects, and a pnd field corresponding to these particles. The methods scat_calc and pnd_calc are called by the parent Cloud object

4.2.2 Droplet

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_scat.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

4.2.3 Gamma

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_scat.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

4.2.4 MonoCrystal

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

4.3 Selected functions

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'

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*, array of particle radii; *CWC*, cloud water content gm^{-3}; *r_eff*, effective radius; *g* (γ), shape parameter

clouds.mh97 (*IWC, r, TK*): MH97 McFarquhar and Heymsfield 1997's particle size distribution **Usage**: n,integrated_n=mh97(IWC,r,TK). **Arguments**: *IWC*, ice water content in gm^{-3}; *r*, Array of radii in microns; *TK*, temperature in Kelvin; *n*, number density in $\mathrm{l}^{-1}\mathrm{mm}^{-1}$ (or $\mathrm{m}^{-3}\mu\mathrm{m}^{-1}$); *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

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

, and the large particle lognormal component is

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

For details see [McFarquharHeymsfield97].

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 $\mathrm{l}^{-1}\mathrm{mm}^{-1}$ (or $\mathrm{m}^{-3}\mu\mathrm{m}^{-1}$)

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

, where

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

,and $B = \frac{c_1}{c_2 r_c^{c_2}}$ **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).

4.4 Algorithm Description and Theoretical Basis

4.4.1 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...

clouds.Crystal.scat_calc (*self, f_grid, za_grid* =[0 10 20 ...], *aa_grid* =[0 10 20 ...], *num_proc* =1): produces npoints scattering data objects, and returns a list (length=npoints) of scattering data files

clouds.Crystal.pnd_calc (*self, LWC_field, IWC_field, T_field*): Returns a list (length=npoints) of pnd fields (GriddedField3), given LWC, IWC and temperature fields (all GriddedField3)

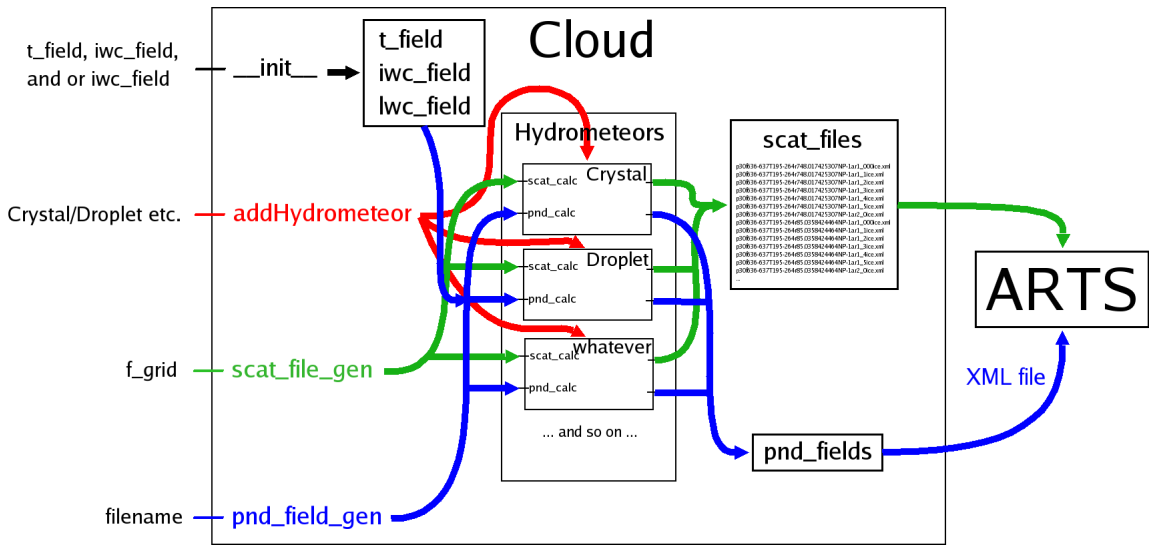This system should allow for the straightforward implementation of new user defined microphysical regimes.

Figure 1: The Cloud class.

4.4.2   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

$$\int_0^\infty n(r)\Phi(r)dr \quad = \quad \int_0^\infty \frac{n(r)}{x^a\exp(-x)}x^a\exp(-x)\Phi(r)\frac{dr}{dx}dx \qquad (2)$$

$$\cong \quad \sum_{i=1}^N \frac{w_i n(r_i)}{x_i^a\exp(-x_i)}\left(\frac{dr}{dx}\right)_i \Phi(r_i)$$

, 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

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

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

4.4.3   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.

4.4.4   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.

4.4.5   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).
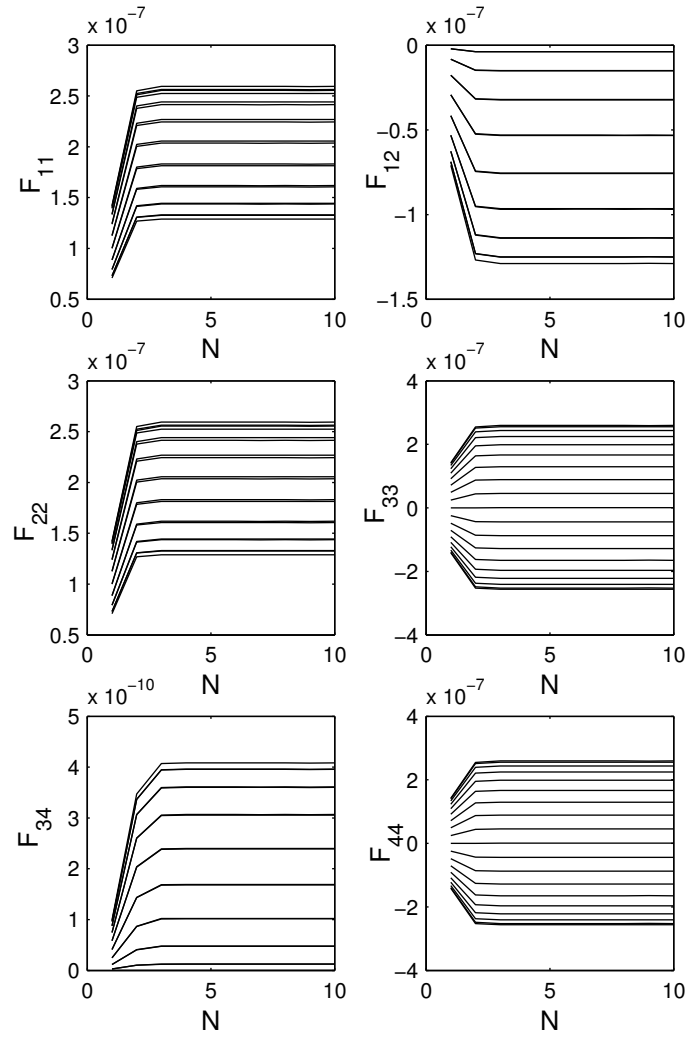
Figure 2: The scattering matrix for liquid spheres, with a liquid water content of 1 gm^{-3}, using a modified gamma size distribution. The x- axes represent the number of quadrature points, and the different lines on each plot are for different scattering angles.

5   The arts_scat module

Implementation of the ARTS Single Scattering Data class. Although this module is used as a back-end to the clouds module, it can be used to easily generate single scattering properties for single hydrometeors.

This example generates single scattering properties for a randomly oriented ice spheroid and saves them in an ART XML File

---

```
>>> from PyARTS import arts_scat

>>> #define the scattering parameters
>>> scat_params={'ptype':20,'equiv_radius':300,'aspect_ratio':0.3,
                  'f_grid':[230e9,240e9],'T_grid':[220,250],'NP':-1,
                  'phase':'ice'}

>>> #show off by doing everything in one line
>>> arts_scat.SingleScatteringData(scat_params).calc().save('a_scattering_data_file.xml')
```

If you wanted to manipulate the data you would do something like ...

```
>>> a=arts_scat.SingleScatteringData(scat_params).calc()
```

...and the scattering properties are in a.pha_mat_data, a.ext_mat_data and a.abs_vec_data.

5.1   SingleScatteringData objects

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 arts_scat 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).

5.1.1   Selected methods

arts_scat.SingleScatteringData.__init__ (*self*, *params* ={}): A SingleScatteringData object is initialised with a dictionary of {*keyword*:*value*} parameters. If this dictionary is omitted, or if any required parameters are missing from the dictionary, default parameters are used. *This is a little dangerous - I might disable the defaults.* After initialisation these parameters (eg: *equiv_radius*, *aspect_ratio*,...) can be changed by modifying the **params** member dictionary, although I prefer to create a new SingleScatteringData object for each set of parameters.

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

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

arts_scat.SingleScatteringData.load (*self*, *filename*, *parse_params* =False): 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 won't be consistent with the *params* data member, unless you specify the optional parse_params argument to be True (default = False). This will extract the params dictionary that was printed in the description field when the file was created This will only work with a file that was created with the arts_scat module
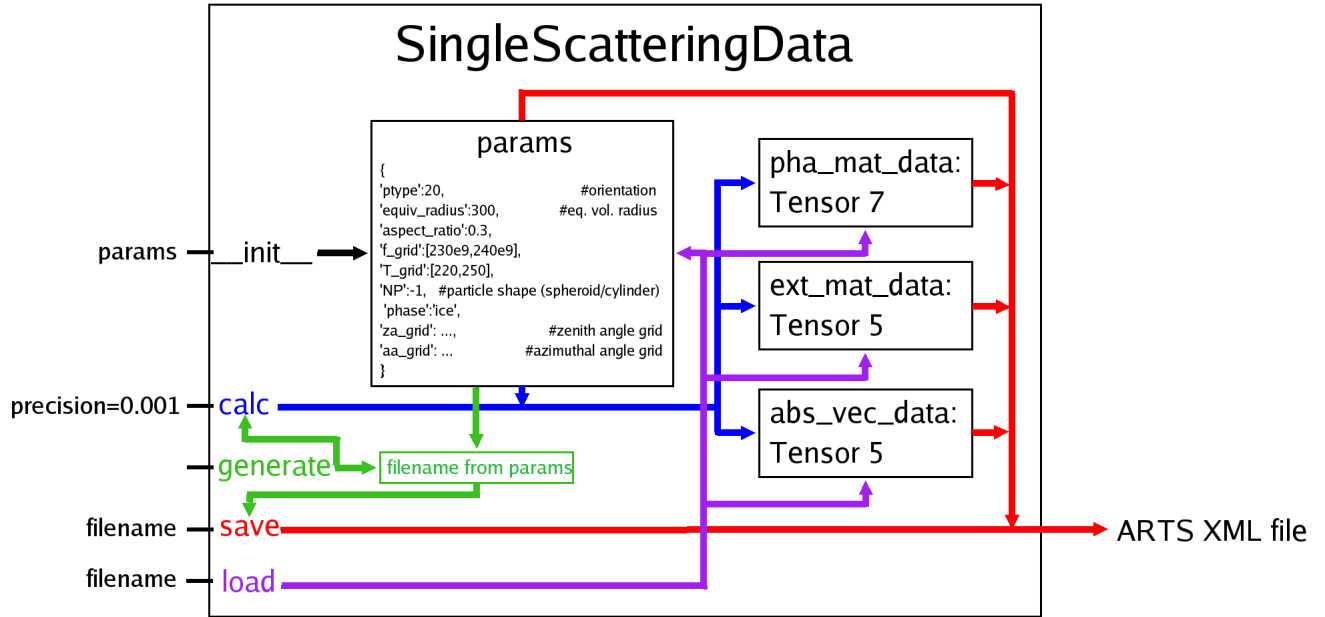
Figure 3: The SingleScatteringData class.

arts_scat.SingleScatteringData.save (*self*, *filename*): Writes single data to <filename> in an arts readable XML format

5.2   Selected functions

arts_scat.batch_generate (*argdict*, *num_proc*): This function takes a dictionary with keys: ['T_grid', 'aa_grid', 'NP', 'equiv_radius', 'aspect_ratio', 'f_grid', 'ptype', 'za_grid'] (just like the SingleScatteringData parameters). However in this case the values for ['T_grid', 'NP', 'equiv_radius', 'aspect_ratio', 'f_grid','ptype'] are lists. batch_generate cascades through these settings generating scattering data files for each combination. The order of calculation is according to the hierarchy: ['f_grid','T_grid', 'ptype', 'NP', 'aspect_ratio', 'equiv_radius'] with 'equiv_radius' changing the fastest. These calculations can be done in num_proc parallel processes. A list of filenames (with an order corresponding to the above hierarchy) is returned

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

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

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

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

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

arts_scat.phasmat (*LAM, THET0, THET, PHI0, PHI, BETA, alpha*): Calculates the phase matrix and returns it in m². This requires that the Tmatrix has already been calculated. See arts_scat.extmat for argument descriptions

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 m². To call this method you must first call tmat_fxd

5.3 Algorithm and Theoretical Basis

5.3.1 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.

5.3.2 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.

5.3.3 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 $\beta = \pi/2$.

5.3.4   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 [mischenko91]. 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.

5.3.5   Calculation of the absorption coefficient vector

Calculation of the absorption coefficient vector is the most taxing part of the arts_scat.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.

5.3.6   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.
    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.

5.4   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.
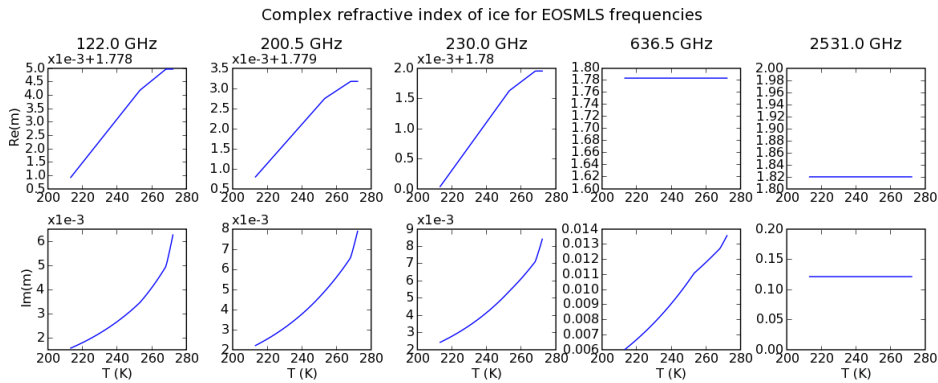
Figure 4: Output of examples/plot_refr_ind.py

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.

6   The `arts_types` module

the arts_types module includes support for the ARTS 1.1.* classes:

- GriddedField3

- ArrayOfGriddedField3

- GasAbsLookup

These classes have the same physical mean as in ARTS. This module allows the generation, manipulation, and input/output in ARTS XML format of these objects

6.1   `GriddedField3` objects

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

6.1.1   Methods

arts_types.GriddedField3.__init__ (*self*, *p_grid* =None, *lat_grid* =None, *lon_grid* =None, *data* =None): GriddedField3 objects are initialised with a pressure grid vector, a latitude vector, a longitude vector and a Tensor3 for the data itself

arts_types.GriddedField3.__call__ (*self*, *p_grid*, *lat_grid*, *lon_grid*): interpolate the field onto new grids, returns only the field data

arts_types.GriddedField3.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

arts_types.GriddedField3.pad (*self*, *plims* =[110000.0, 1 ...], *latlims* =[-90, 90], *lonlims* =[-180, 180]): Adds extra gridpoints at new extremities in all dimensions. data values are copied from the existing end points

arts_types.GriddedField3.load (*self*, *filename*): load a GriddedField3 object from and ARTS XML file

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

6.2   `GasAbsLookup` objects

This class enables the calculation of GasAbsLookup tables for arts 1.1.* using the stable branch arts 1.0.*. An example of using this class will soon be provided.

6.2.1   Methods

arts_types.GasAbsLookup.__init__ (*self*, *species* =None, *f_grid* =None, *p_grid* =None, *T_pert* =None): The initialisation arguments are *species*, a list of lists of species tags as appears in the arts 1.1 GasAbsLookupTable; and vectors *f_grid*, *p_grid*, and *T_pert*, representing frequency, pressure and temperature perturbations

arts_types.GasAbsLookup.__call__ (*self*, *f_index*, *pressure*, *temperature*, *vmrs*): return a vector containing the contribution of each species to the scalar gas absorption coefficient. Before this can be called the method set_slidata must be called (only once). This is analagous to GasAbsLookup::Extract in ARTS

arts_types.GasAbsLookup.calc (*self*, *tags*, *hitran_filename*, *line_fmin*, *line_fmax*, *atm_basename*):
Calculated the lookup table using arts 1.0. **Input:** *tags*, list of arts 1.0 tags e.g. ['H2O,H2O-MPM93','O2,O2-MPM93']; *hitran_filename*, the name of the hitran line file; *line_fmin*, float - the minimum frequency line to consider; *line_fmax*, float - the maximum frequency line to consider; *atm_basename*, string - the basename where the atmospheric profiles can be found in arts1.0 ascii format

arts_types.GasAbsLookup.load (*self*, *filename*): loads gas abs lookup table from an ARTS 1.1. XML file

arts_types.GasAbsLookup.save (*self*, *filename*): Saves the lookup table in arts-1.1 XML format

arts_types.GasAbsLookup.set_slidata (*self*): Sets the sli_data member, which is used by the __call__ function to interpolate the lookup table.

7   The `plotting` module

General purpose plotting functions using the matplotlib package.

7.1   `SentinelMap` objects

SentinelMap is a matplotlib colormap that deals with data points that you want to distinguish from the rest of the data. For example if bad data is stored as -999, these values are plotted a specified rgb color, and the rest of the colormap (cmap) is unchanged. This needs to be used with the SentinelNorm class. e.g.

```
>>> cmap = SentinelMap(cm.jet, -999, (0,0,0))
>>> norm = SentinelNorm(-999)
>>> pcolor(x,y,z,norm=norm,cmap=cmap)
```

will plot the data with the usual jet colormap but with bad data values (-999) black.

7.2   Selected functions

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

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

plotting.mySubplot (*nrows*, *ncols*, *pnum*, *figpos* =[0.050000000 ...], *axpos* =[0.149999999 ...]): 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.

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

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

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

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

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

plotting.shiftaxes (*ax*, *delta_pos*): For use with matplotlib. **input:** *ax*, a matplotlib.axes object; *delta_pos*, a 4 element list or array correspond to [delta_x_start,delta_y_start,delta_width,delta_height] in normalised units.

7.2.1   Demonstration

Figure 4 shows the output of examples/geometry.py, which uses . `plotting.drawCloudBox`, `plotting.drawPpath`, and `plotting.drawSurface`.
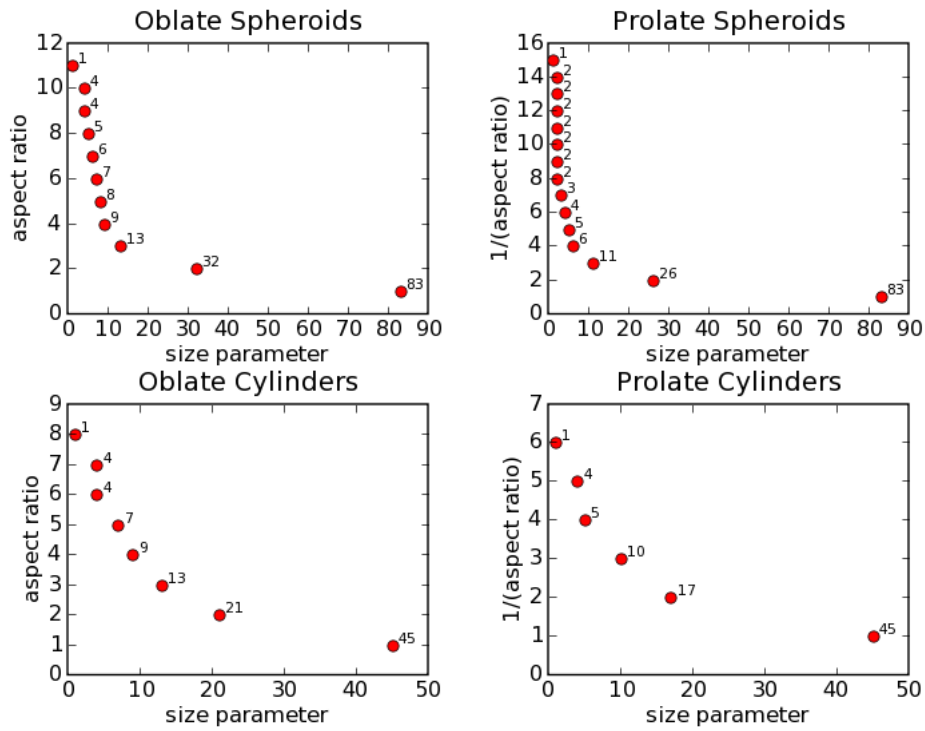
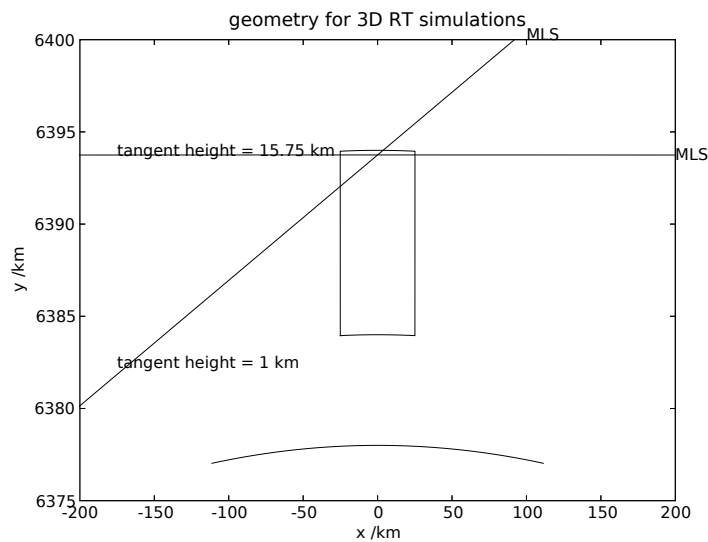Figure 5: *T*-matrix convergence failure parameters for m=(1.78117084503+0.00504761422053j)

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

8 The `artsXML` module

The artsXML module deals with the creation and loading of data files in the ARTS XML format.
For XML output, the main class is XMLfile. An XMLfile object is initialised with a filename.

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

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

```
>>> a_tensor=ones([3,5,6],Float)
>>> testfile.addTensor(a_tensor)
```

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()
```

Some shortcut save*** functions are available. Using saveTensor the above is achieved in one line.

```
>>> artsXML.saveTensor(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 general purpose function that returns a dictionary structure reflecting the structure of the XML file. As far as far as I know, this works with every data type exported by ARTS.

Elsewhere in the PyARTS package there are classes with load methods for more specialised types such as GriddedField3, or SingleScatteringData. All of these will use artsXML.load as a backend.

8.1 `XMLfile` objects

arts XML output class. Initialise with a filename

8.1.1 Selected Methods

artsXML.XMLfile.addArray (*self, arraylist*): takes a list of python arrays of the same rank and writes an XML ArrayOfTensor<n>n structure

artsXML.XMLfile.addArrayOfArray (*self, data*): takes a list of lists of python arrays of the same rank and writes an XML ArrayOfArrayOfTensorn structure

artsXML.XMLfile.addArrayOfString (*self, data*): takes an array of Strings and writes an ArrayOfTensor structure

artsXML.XMLfile.addString (*self, data*): adds a String

artsXML.XMLfile.addTensor (*self, tensor*): This method takes a numpy array argument and stores it in the arts XML format with the appropriate Tag (eg Vector Matrix , ...

artsXML.XMLfile.close (*self*): This must be called to finalise the XML file

8.2   Selected functions

artsXML.saveTensor (*data*, *filename*): saves a python array in the appropriate arts xml format (eg Vector, Matrix etc)

artsXML.saveArray (*data*, *filename*): saves a list of python arrays in the appropriate arts xml format (eg ArrayOfVector,ArrayOfMatrix etc)

artsXML.saveString (*data*, *filename*): saves Strings in the appropriate arts xml format (e.g. String or ArrayOfString)

artsXML.load (*filename*, *use_names* =True): 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 Usage: data_struct=load(filename)

9   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.

9.1   Selected functions

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

arts_math.multi_gauss_leg (*func*, *rangelist*, *n* =10): arts_math.multi_gauss_leg has no docstring!

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

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

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)

10   The general module

This file includes interpreter or general purpose functions

10.1   Selected functions

general.multi_thread (*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)

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)

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

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.

11   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.

11.1   `SLIData2` objects

Class for 2D sequential linear interpolation

11.1.1   Selected methods

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

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.

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

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

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

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

11.1.2   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).
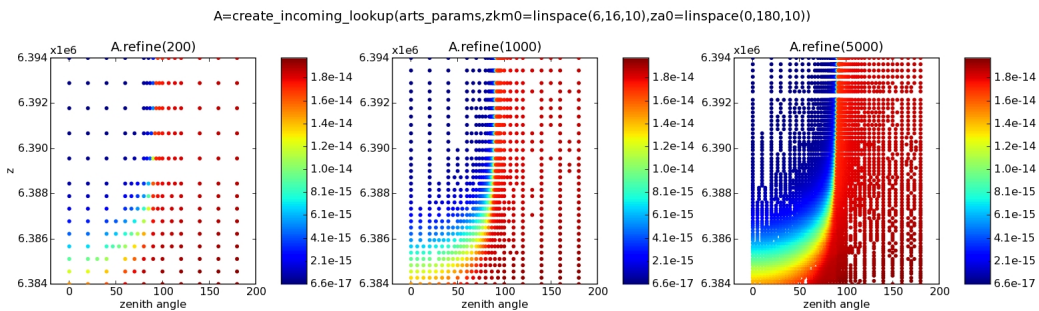

Figure 7: the output of examples/mc_incoming_gen.py, which makes use of the `SLIData2 class`.

12   The `arts1` module

This module deals with control file creation and data input/output for the stable arts 1 package. This is used only for the creation of GasAbsLookup tables. To use this module you need to set the envoronment variable ARTS1_PATH or manually set arts1.ARTS1_EXEC after importing this module

12.1   Selected functions

arts1.abs_file (*tags*, *hitran_filename*, *line_fmin*, *line_fmax*, *atm_basename*, *p_file*, *T_offset*, *f_file*): creates a command list corresponding to the calculation of absorption coefficients for specified 1D atmospheric fields. This function is used by the GasAbsLookup class

arts1.arts1_file_from_command_list (*command_list*, *filename*): takes a list of WSM and AgendaSet objects and creates an arts control file

arts1.arts1_load (*filename*): load an array aor a list of arrays from an arts 1 ascii file

arts1.arts1_save (*x*, *filename*): save a numpy array (1D or 2D) in arts 1 ascii format

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 nonspherical 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.

Generated on: 2006-09-20. Generated by Docutils from reStructuredText source.