ARTS User Guide

edited by

Patrick Eriksson[1] **and Stefan Bühler**[2]

March 23, 2009
ARTS Version 1.11.50

The content and usage of ARTS are not only described by this document. An overview of ARTS documentation and help features are given in Section 1.2. For continuous reports on changes of the source code and this user guide, subscribe to the ARTS developers mailing list at `http://www.sat.ltu.se/arts/support/`.

We welcome gladly comments and reports on errors in the document. Send then an e-mail to: `patrick (at) rss.chalmers.se` **or** `sbuehler (at) ltu.se`**.**

If you use data generated by ARTS in a scientific publication, then please mention this and cite the most appropriate of the ARTS publications that are summarized on `http://www.sat.ltu.se/arts/docs/`.

[1]Department of Radio and Space Science, Chalmers University of Technology, SE-41296 Göteborg, Sweden
[2]Department of Space Science, Luleå University of Technology, Box 812, SE-98128 Kiruna, Sweden

Contributing authors

Author/email	Main contribution(s)
Stefan Bühler[d] sbuehler (at) ltu.se	Editor, Sections 2, 6, 15, 16, 17 and 18.
Cory Davis[c] cory (at) met.ed.ac.uk	Section 14.
Mattias Ekström[b] ekstrom (at) rss.chalmers.se	Section 10, 17.6.
Claudia Emde[a] claudia.emde (at) dlr.de	Sections 5.1, 12, 13, 20 and 21.
Patrick Eriksson[b] Patrick.Eriksson (at) rss.chalmers.se	Editor, report structure, Sections 3, 4, 7, 8, 9, 11 and 21.
Oliver Lemke[d] olemke (at) core-dump.info	Latex fixes and automatic generation of appendices.
Christian Melsheimer[a] cmels (at) sat.physik.uni-bremen.de	Section 22.
Sreerekha T.R.[a] rekha (at) sat.physik.uni-bremen.de	Section 19.

[a] Institute of Environmental Physics, University of Bremen,
P.O. Box 33044, D-28334 Bremen, Germany

[b] Department of Radio and Space Science, Chalmers University of Technology,
SE-41296 Göteborg, Sweden

[c] Institute for Atmospheric and Environmental Science, University of Edinburgh,
EH93JZ Edinburgh, Scotland, UK

[d] Department of Space Science, Luleå University of Technology,
Box 812, SE-98128 Kiruna, Sweden

Contents

Part I

Overview

Chapter 1

Introduction

Some nice welcome text ...

1.1 Temporary internal notes

Below you find a list of things to do. Please add and remove items when appropriate. Let's try to keep this list rather complete and to have a person assigned for each point. This in order to keep up the speed of creating this user guide and that no point is continously expected to be fixed by someone else.

- Write this section. [Stefan/Patrick]
- Section 2 is marked as under construction. Correct? Update or remove comment. [Stefan]
- Something shall be written about polarisation and Stokes in Section 3.3. [Christian/Claudia]
- Introduction to absorption and refractive index has to be written. The corresponding chapter in Part 1 is empty. [???]
- Section 3.5.3 on Sensor polarisation to be written. Finer details shall be put in a chapter on Sensor characteristics in Part 1. [Patrick/Christian]
- The theory we use for radiative transfer shall be described in Part IV in as general terms as possible. The start of the scattering chapter should be moved to the new chapter. The case without scattering can then be described as a special case of the general expression. To gather the description of radiative transfer should be better than to describe it in parts at different places in the user guide. [Claudia/Christian]
- An introductory part on scattering calculations shall be found in Sec. 3.8 [Claudia/Shreerekha]

History
02xxxx xxx.

- Start a chapter on clear sky radiative transfer for Part I. [Patrick]
- Is scattering chapter OK? [Claudia/Shreerekha]
- Fix chapter on agendas. [Stefan]
- There are some FIXME in chapter on Polarisation and Stokes Parameters. [Christian]
- In the appendix for WSM: The method names appear strange (starts with levelb). The list of input and output varaibles needs a row break. [Oliver]

1.2 Documentation guide

Describe where different type of information can be found. For example, refer to full control file examples in `tests/`.

1.3 Background

The number of satellite sensors in the millimeter and sub-millimeter spectral range is rapidly growing. They use various frequency bands and observation geometries. Two important groups of sensors are for example the nadir viewing millimeter wave sensors like AMSU[1] and the limb viewing sub-millimeter wave sensors like the planned SMILES[2].

For the data analysis all such sensors require accurate and fast forward models, which can simulate measurements for a given atmospheric (and maybe ground) state. Depending on the objective of the sensor, the measurement will depend for example on the distribution of atmospheric temperature, water vapor, ozone, and many other trace gases.

So far, a lot of effort has been wasted in developing dedicated forward models for different sensors, although all these models have many features in common. Moreover, existing models were not easily modifiable and extendable. Hence, it was decided to develop a new model which emphasizes modularity, extensibility, and generality.

[* Describe how ARTS was initiated and started. Release of version 1. *]

1.4 What is ARTS

[* ??? *]

1.5 The scope of ARTS

[* Update old text and add new stuff. *]

[1] The **A**dvanced **M**icrowave **S**ounding **U**nit is a sensor on board the polar orbiting satellites of the US-American National Aeronautics and Space Administration.

[2] The **S**uperconducting Sub-**Mi**llimeter Wave **L**imb **E**mission **S**ounder is a Japanese Sensor which will be flown for the first time on the International Space Station.

1.6 Additional tools

[* Update old text and add new stuff. *]

Chapter 2

ARTS: concept and the programme

This section describes the basic ideas underlying the ARTS programme. It also introduces some terminology. You should read it if you want to understand how the program works and how it can be used efficiently.

This section is not about physics, only about ARTS as a computer program. Refer to Section 3 for an introduction to the physics of atmospheric radiative transfer and its mathematical description in ARTS.

2.1 Main components

The most important notion in ARTS is the workspace. All physical quantities (for example absorption coefficients) are workspace variables. But workspace variables can also be of a more technical nature, for example various grids.

The program performs a calculation by executing a list of workspace methods, which are specified in a controlfile. These workspace methods take workspace variables as input, and generate workspace variables as output.

It is important to note that the controlfile has a fixed and well-defined syntax. This syntax is understood by the ARTS parser. The great advantage of this concept is that it is very easy to add new workspace variables and new workspace methods. The program has an internal lookup table which lists all workspace methods, as well as their input variables, output variables, and generic input/output parameters. To add a new method, one just has to add an entry to this lookup table, and write the code for the method itself. No further changes to the program are necessary. In particular, no changes to the program logic or to the parser. How such an extension can be made practically is described in Section 15.

History

080729	Section on command line parameters updated by Stefan Buehler.
050613	Updated by Patrick Eriksson.
020613	Updated and extended by Stefan Buehler.
000616	Created by Stefan Buehler, based on my DPG2000 poster.

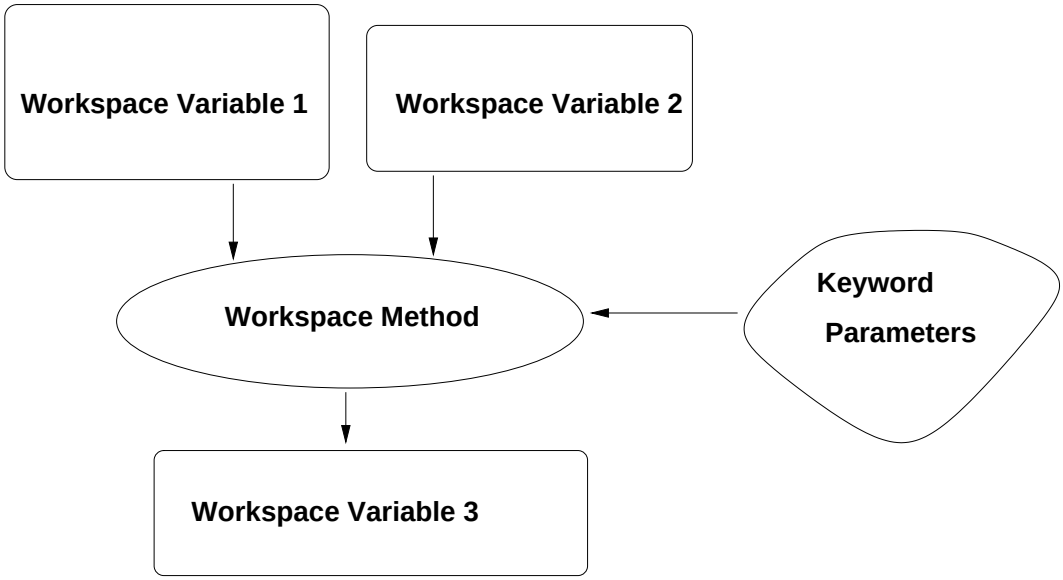

Figure 2.1: Specific workspace methods act on specific workspace variables to generate other specific workspace variables. Additional input parameters can be specified as generic input and output parameters in the controlfile.

2.2 Generic workspace methods

Generic methods (Figure 2.2) allow the user of the program even more freedom than specific methods. A generic method is for example `MatrixSetConstant`, which can be used to set any workspace variable which is a matrix. For example

```
MatrixSetConstant( z_surface, 10, 10, 0.0 )
```

will set all elements of `z_surface` to 0.0 (as long as `nrows` and `ncols` are set).

Some methods are even more flexible, the are super generic. This means that they can take any workspace variable as input. The most commonly used such methods are the XML file methods. A workspace variable is read from a file in this way

```
ReadXML(f_grid){"frequency_grid"}
```

Generic methods are particularly useful for IO operations like in the example above. No new IO methods are necessary for new workspace variables, as long as they are of standard types already known to the program (for example vectors or matrices).

2.3 Agendas

Agendas are a special incarnation of a workspace method. In the controlfile an arbitrary number of workspace methods can be added to an agenda. On invocation, the agenda executes its methods one after the other. The inputs and outputs defined for the agenda must be satisfied by the invoked workspace methods. E.g., if an agenda has `f_grid` in its list of

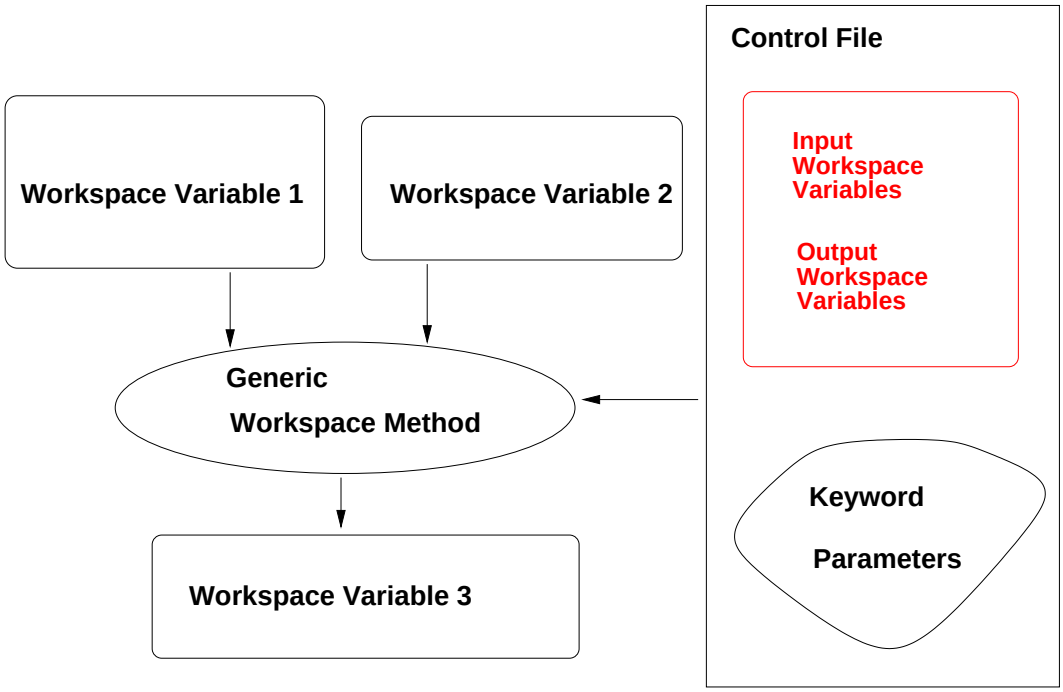

Figure 2.2: For generic workspace methods the workspace variables to act on are specified in the controlfile.

output workspace variables, a workspace method which generates `f_grid` must be added to the agenda in the controlfile.

Even though it is possible to execute agendas directly from the controlfile with the `AgendaExecute` method, the more common and intended use case is the internal invocation by other workspace methods. This adds a grave amount of flexibility to arts. The `RteStd` method for example calculates (besides other components) the emission term. Without the means of an agenda, it would only be possible to use always the same method for the emission calculation. By the use of an agenda the user can choose between different methods to calculate the emission and plug them into the emission agenda in the control file:

```
AgendaSet( emission_agenda ){
  emissionPlanck
}
```

`RteStd` internally calls the `emission_agenda` and uses the user selected method for calculating the emission term.

2.4 Practical hints

2.4.1 Test controlfiles

The subdirectory `tests` contains some example controlfiles. You should study them to learn more about how the program works. You can also run these controlfiles like this:

```
arts TestAbs.arts
```

This assumes that you are inside the directory where the controlfiles are, and that the `arts` executable is in your path. You can also run all of the examples, by saying

```
make check
```

2.4.2 Command line parameters

ARTS offers a number of useful command line parameters. In general, there is a short form and a long form for each parameter. The short form consists of a minus sign and a single letter, whereas the long form consists of two minus signs and a descriptive name.

Help

To get a full list of available command line parameters, type

```
arts -h
```

or

```
arts --help
```

Online documentation

Most useful at the beginning should be the `-d` (`--describe`), `-m` (`--methods`), `-w` (`--workspacevariables`), and `-i` (`--input`) flags. For instance, the `-d` (`--describe`) flag gives you online documentation for any workspace method or workspace variable. Usage:

```
arts -d f_grid
```

will print documentation about the workspace variable `f_grid`, which happens to be the monochromatic frequency grid.

But what methods and variables are available? You can find out by typing

```
arts -m all
```

which will list all workspace methods, or by typing

```
arts -w all
```

which will list all workspace variables. As you can see, these lists are quite long. But you can get more specific information:

```
arts -m f_grid
```

will give you a list of all methods that can generate the workspace variable `f_grid`. Specific and generic methods are listed separately. Generic methods are in this case all methods producing a Vector, since `f_grid` belongs to this group. A similar task is performed by the `-i` (`--input`) flag, with the difference that `arts -i f_grid` will list those methods that require `f_grid` as *input*, whereas `arts -m f_grid` lists those that produce `f_grid` as output. Finally,

```
arts -w surfaceFlat
```

will give you all variables required by the method `abs_coefCalc` (the variable `f_grid` happens to be one of them).

Using these command line parameters, it is easy to build up a controlfile. The trick is, to start at the end. Say you want to compute absorption coefficients. First of all, you have to find out in which workspace variable these are stored. Look at the list produced by `arts -w all`. You can use `arts -d` to look at some candidates a bit more closely. This way, you will find out that `abs_coef` is the variable you are looking for.

In the next step, you can use `arts -m abs_coef` to find all methods that can calculate `abs_coef`. So, you will find the method `abs_coefCalc`. Now you can use `arts -w abs_coefCalc` to find out the required input variables of that method. Then you can use the `-m` flag again, to find the methods producing these variables, and so on.

Verbosity levels

The command line parameter

```
arts -r
```

or

```
arts --reporting
```

can be used to set how much output ARTS produces. You can supply a three-digit integer here. Each digit can have a value between 0 and 3.

The last digit determines, how verbose ARTS is in its report file. If it is 0, the report file will be empty, if it is 3 it will be longest.

The middle digit determines, how verbose ARTS is on the screen (stdout). The meaning of the values is exactly as for the report file.

The first digit is special. It determines how much you will see of the output of agendas (other than the main program agenda). Normally, you do not want to see this output, since many agendas are called over and over again in a normal program run.

The agenda verbosity applies in addition to the screen or file verbosity. For example, if you set the reporting level to '123', you will get:

- From the main agenda: Level 1-2 outputs to the screen, and level 1-3 outputs to the report file.
- From all other agendas: Only level 1 outputs to both screen and report file.

If you set the reporting level to ‘120’, the report file will be empty.

The default setting for ARTS (if you do not use the command line flag) is ‘010’, i.e., only the important messages to the screen, nothing to the report file, and no sub-agenda output.

Chapter 3

The forward model: concepts, definitions and overview

This chapter introduces terms and concepts of ARTS as a forward model, in contrast to the previous chapter that describes ARTS as a computer program. While the content of the previous chapter is specific for ARTS, as the way to use a forward model program normally differs significantly from one implementation to another, this chapter is of more general nature. Most of the quantities treated here should be part of any forward model of the same complexity as ARTS, where only details regarding the definition should differ. The aim of this chapter is to give an overview of the forward model and to describe important terms and concepts, in such a way that the content of this user guide can be fully appreciated and that you shall understand how to construct a control file for your simulation problem.

3.1 The atmosphere

3.1.1 Atmospheric dimensionality

The structure of the modelled atmosphere can be selected to have different degree of complexity, the atmospheric dimensionality. There exist three levels for the complexity of the atmosphere, 1D, 2D and 3D, where 1D and 2D can be seen as special cases of 3D. The significance of these different atmospheric dimensionalities and the coordinate systems used are described below in this section. The atmospheric dimensionality is selected by setting the workspace variable `atmosphere_dim` to a value between 1 and 3. Variables for which the size depends on the atmospheric dimensionality are checked, when used, to have a size consistent with `atmosphere_dim`. The atmospheric dimensionality is most easily set by the functions `AtmosphereSet1D`, `AtmosphereSet2D` and `AtmosphereSet3D`.

3D In this, the most general, case, the atmospheric fields vary in all three spatial coordinates, as in a true atmosphere (Figures 3.3 and 3.4). A spherical coordinate system

---

History

050613 First complete version finished by Patrick Eriksson.

020315 Started by Patrick Eriksson.

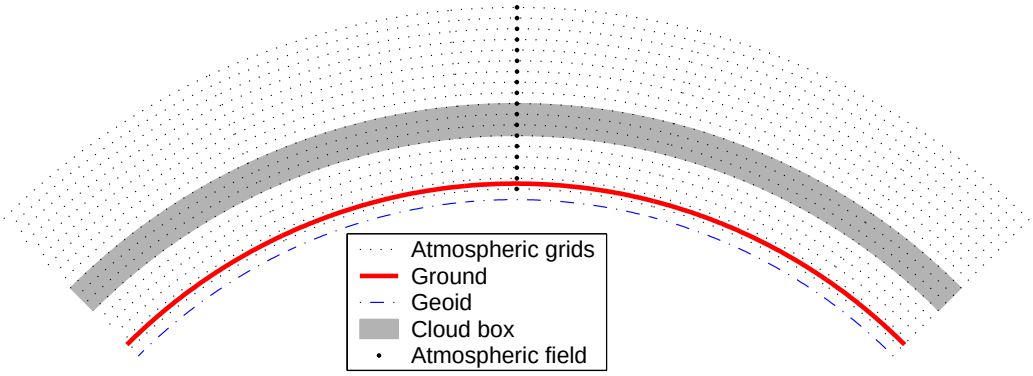

Figure 3.1: Schematic of a 1D atmosphere. The atmosphere is here spherically symmetric. This means that the radius of the geoid, the surface and all the pressure levels are constant around the globe. The fields are specified by a value for each pressure level. The extension of the cloud box is either from the surface up to a pressure level, or between two pressure levels (which is the case shown in the figure). The figure shows further that the surface must be above the lowermost pressure level.

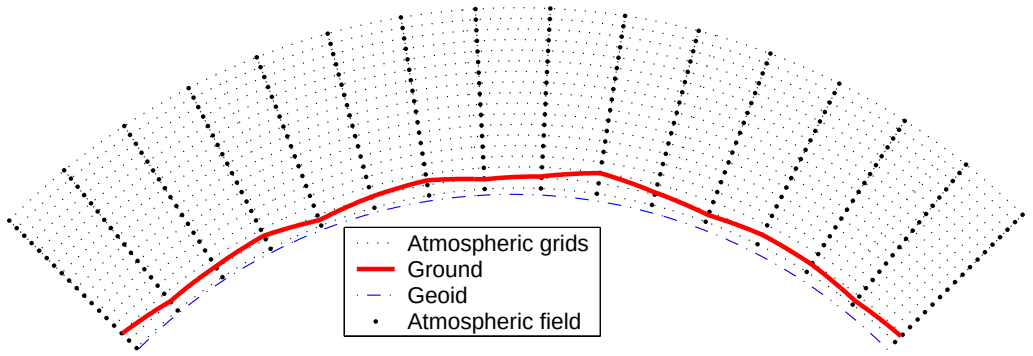

Figure 3.2: Schematic of a 2D atmosphere. The radii (for the geoid, the surface and the pressure levels) vary here linear between the latitude grid points. The atmospheric fields vary linearly along the pressure levels and the latitude grid points (that is, along the dotted lines). Inside the grid cells, the fields have a bi-linear variation. The cloud box is not defined for 2D.

is used where the dimensions are radius (r), latitude (α) and longitude (β), and a position is given as (r, α, β). With other words, the standard way to specify a geographical position is followed. However, the way to specify the radial position differs depending on the context, which is described in Section 3.1.2. The valid range for latitudes is $[-90°, +90°]$, where +90°corresponds to the North pole etc. Longitudes are counted from the Greenwich meridian with positive values towards the east. Longitudes can have values from -360°to +360°. When the difference between the last and first value of the longitude grid is $\geq 360°$ then the whole globe is considered to

be covered. The user must ensure that the atmospheric fields for β and $\beta + 360°$ are equal. If a point of propagation path is found to be outside the range of the longitude grid, this will results in an error if not the whole globe is covered. When possible, the longitude is shifted with 360° in the relevant direction.

1D A 1D atmosphere can be described as being spherically symmetric (Figure 3.1). The term 1D is used here for simplicity and historical reasons, not because it is a true 1D case (a strictly 1D atmosphere would just extend along a line). A spherical symmetry means that atmospheric fields and the surface extend in all three dimensions, but they have no latitude and longitude variation. This means that, for example, atmospheric fields vary only as a function of altitude and the surface constitutes the surface of a sphere. The radial coordinate is accordingly sufficient when dealing with atmospheric quantities, but the angular distance between the sensor and a point along the propagation path can be of interest, for example when determining the cross-link between two satellites (a fact that shows that this is not a true 1D case). A polar coordinate system is for this reason used when describing propagation paths, where the coordinate additional to the radius gives the angular distance inside the viewing plane between the sensor and the point of interest (see also Section 7.3). This latter coordinate system coincides with the one used for 2D if the sensor position is set to be the zero point for the latitudes. A 1D atmosphere is shown in Figure 3.1.

2D In contrast to the 1D and 3D cases, a 2D atmosphere extends only inside a plane (Figure 3.2). A spherical coordinate system is accordingly not needed and a polar system, consisting of a radial and an angular coordinate, is used. The 2D case is most likely used for satellite measurements where the atmosphere is observed inside the orbit plane. The angular coordinate corresponds then to the angular distance along the satellite track, but the coordinate is for simplicity denoted as the latitude. The zero point for the 2D latitude is arbitrary. No lower and upper limit exists for the 2D latitude, and this allows that measurements from several subsequent orbits can be simulated as one unit. The atmosphere is treated to be undefined outside the considered plane. A 2D atmosphere is shown in Figure 3.2.

3.1.2 Altitude coordinates

Pressure The main altitude coordinate is pressure. This is most clearly manifested by the fact that the vertical atmospheric grid consists of levels with equal pressure. The vertical grid is consistently denoted as the pressure grid and the corresponding workspace variable is `p_grid`. The choice of having pressure as main altitude coordinate results in that atmospheric quantities are retrieved as a function of pressure, not as a function of geometrical altitude.

Pressure altitude A basic assumption in ARTS is that atmospheric quantities (temperature, geometric altitude, species VMR etc.) vary linearly with the logarithm of the pressure. This corresponds roughly to assuming a linear variation with altitude.

Radius Geometrical altitudes are needed to determine the propagation path through the atmosphere etc. The main geometrical altitude coordinate is the distance to the centre

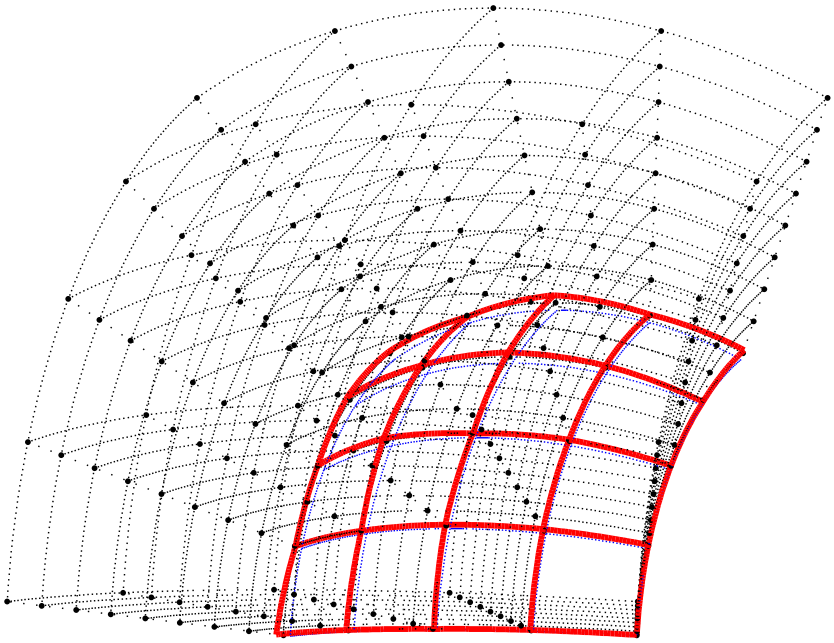

Figure 3.3: Schematic of a 3D atmosphere. Plotting symbols as in Figure 3.2. Radii and fields are here defined to vary linearly along the latitude and longitude grid points. This means that surfaces (such as the radius of the surface) have a bi-linear variation inside the area limited by two latitude and longitude grid values, while the atmospheric fields have a tri-linear variation inside the grid cells.

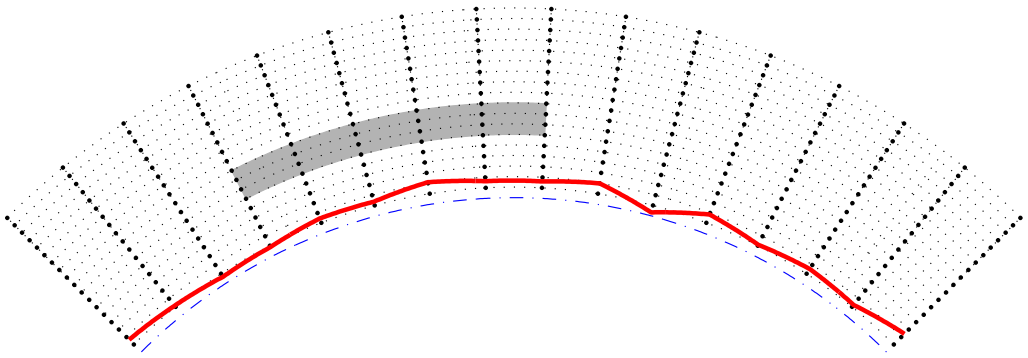

Figure 3.4: A latitudinal, or longitudinal, cross section of a 3D atmosphere. Plotting symbols as in Figure 3.1. Radii and fields inside the cross section match the definitions for 2D. The vertical extension of the cloud box is defined identical for 1D and 3D. The horizontal extension of the cloud box is between two latitude and longitude grid positions, where only one of the dimensions are visible in this figure.

of the coordinate system used, the radius. This is a natural consequence of using a spherical or polar coordinate system. The radius is used inside ARTS for all geometrical calculations and to store the position of the sensor (Section 3.4).

Geometrical altitude The term geometrical altitude signifies here the difference in radius between a point and the geoid (Section 3.1.4) along the vector to the centre of the coordinate system (Equation 3.1). Hence, the geometrical altitude is not measured along the local zenith direction (the normal to the reference geoid). Geometrical altitudes are mainly used to facilitate the input of the surface altitude, the altitude of the pressure levels etc. This is the case as these quantities are known rather with respect to the geoid than with respect to the Earth's centre.

3.1.3 Atmospheric grids and fields

As mentioned above, the vertical grid of the atmosphere consists of a set of layers with equal pressure, the pressure grid (`p_grid`). This grid must of course always be specified. It is not allowed that there is an altitude gap between the surface and the lowermost pressure level. That is, the surface pressure must be smaller than the pressure of the lowermost vertical grid level. On the other hand, it is not necessary to match the surface and the first pressure level, the pressure grid can extend below the surface level. The upper end of the pressure grid gives the practical upper limit of the atmosphere as vacuum is assumed above. With other words, no absorption and refraction take place above the uppermost pressure level.

A latitude grid (`lat_grid`) must be specified for 2D and 3D. For 2D, the latitudes shall be treated as the angular distance along the orbit track, as described above in Section 3.1.1. The latitude angle is throughout calculated for the vector going from the centre of the coordinate system to the point of concern. Hence, the latitudes here correspond to the definition of the geocentric latitude, and not geodetic latitudes (see Section 7.9.1). This is in accordance to the definition of geometric altitudes found above. For 3D, a longitude grid (`lon_grid`) must also be specified. Valid ranges for latitude and longitude values are given in Section 3.1.1.

The atmosphere is treated to be undefined outside the latitude and longitude ranges covered by the grids, if not the whole globe is covered. This results in that a propagation path is not allowed to cross a latitude or longitude end face of the atmosphere, if such exists, it can only enter or leave the atmosphere through the top of the atmosphere (the uppermost pressure level). See further Section 3.5.2. The volume (or area for 2D) covered by the grids is denoted as the model atmosphere.

If the longitude and latitude grids are not used for the selected atmospheric dimensionality, then the longitude grid (for 1D and 2D) and the latitude grid (for 1D) must be set to be empty, but when dealing with the size of variables the grid length shall be treated to be one. For example, the matrix describing the geoid (see Section 3.1.4) has for 1D the size $[1, 1]$.

The basic atmospheric quantities are represented by their values at each crossing of the involved grids (indicated by thick dots in Figures 3.1 - 3.4), or for 1D at each pressure level (thick dots in Figure 3.1). This representation is denoted as the field of the quantity. The field must, at least, be specified for the geometric altitude of the pressure levels (`z_field`), the temperature (`t_field`) and considered atmospheric species (`vmr_field`). The fields are assumed to be piece-wise linear functions vertically (with pressure altitude as the vertical

coordinate, Section 3.1.2), and along the latitude and longitude edges of 2D and 3D grid boxes. For points inside 2D and 3D grid boxes, multidimensional linear interpolation is applied (that is, bilinear interpolation for 2D etc.). Note especially that this is also valid for the field of geometrical altitudes (`z_field`). Fields are rank-3 tensors. For example, the temperature field is $T = T(P, \alpha, \beta)$. That means each field is like a book, with one page for each pressure grid point, one row for each latitude grid point, and one column for each longitude grid point. In the 1-D case there is just one row and one column on each page.

3.1.4 The geoid and the surface

The geoid is an imaginary surface used as a reference when specifying the surface altitude and the altitude of pressure levels. Any shape of the geoid is allowed but a smoothly varying geoid is the natural choice, with the enters of the geoid and the coordinate system coinciding. The geoid should normally be set to the reference ellipsoid for some global geodetic datum, such as WGS-84. For further reading on geoid ellipsoids and WGS-84, see Section 7.9.1.

Inside ARTS, the geoid is represented as a matrix (`r_geoid`), holding the geoid radius, $r_\odot$, for each crossing of the latitude and longitude grids, $r_\odot = r_\odot(\alpha, \beta)$. The geoid is not defined outside the ranges covered by the latitude and longitude grids, with the exception for 1D where the geoid by definition is a full sphere. The surface altitude, z_g, is given as the geometrical altitude above the geoid. The radius for the surface is accordingly

$$r_g = r_\odot + z_g \tag{3.1}$$

As already mentioned, a gap between the surface and the lowermost pressure level is not allowed.

The ARTS variable for the surface altitude (`z_surface`) is a matrix of the same size as the geoid matrix. For 1D, the surface is a sphere by definition (as the geoid), while for 2D and 3D any shape is allowed and a rough model of the surface topography can be made. The treatment of surface emission and reflectivity is discussed in Section 3.5.5.

3.1.5 The cloud box

In order to save computational time, scattering calculations are limited as far as possible to the part of the atmosphere containing clouds and other scattering objects. The atmospheric region in which scattering shall be considered is denoted as the cloud box, and it is discussed here as it acts as an additional atmospheric limit when calculating propagation paths (see Section 3.5). Cloud box calculations can be performed in 1D and 3D mode, but not for 2D.

The cloud box is defined to be rectangular in the used coordinate system, with limits exactly at points of the involved grids. This means, for example, that the vertical limits of the cloud box are two pressure levels. For 3D, the horizontal extension of the cloud box is between two points of the latitude grid and likewise in the longitude direction (Figure 3.4). The latitude and longitude limits for the cloud box cannot be placed at the end points of the corresponding grid as it must be possible to calculate the incoming intensity field. The cloud box is activated by setting the variable `cloudbox_on` to 1. The limits of the cloud box are stored in `cloudbox_limits`. It is recommended to use the method `CloudboxOff`

when no scattering calculations shall be performed. This method assigns dummy values to all workspace variables not needed when scattering is neglected.

When the radiation entering the cloud box is calculated this is done with the cloud box turned off. This to avoid to end up in the situation that the radiation entering the cloud box depends on the radiation coming out from the cloud box. **It is the task of the user to define the cloud box in such way that the link between the outgoing and ingoing radiation fields of the cloud box can be neglected**. The main point to consider here is radiation reflected by the surface. To be formally correct there should never be a gap between the surface and the cloud box. This is the case as radiation leaving the cloud box can then be reflected back into the cloud box by the surface. If it is considered that the surface is a scattering object it is clear that the surface should in general be part of the cloud box. However, for many cases it can be accepted to have a gap between the surface and the cloud box, with the gain that the cloud box can be made smaller. Such a case is when the surface is treated to act as blackbody, the surface is then not reflecting any radiation. Reflections from the surface can also be neglected if the zenith optical thickness of the atmosphere between the surface and cloud box is sufficiently high.

3.2 Stokes dimensionality

To full polarisation state of radiation can be described by the Stokes vector. The vector can be defined in different ways, but it has always four elements. The Stokes vector, $\mathbf{I}$, is here written as

$$\mathbf{I} = \begin{bmatrix} I \\ Q \\ U \\ V \end{bmatrix}, \tag{3.2}$$

where the first component (I) is the full intensity of the radiation, the second component (Q) is the difference between vertical and horizontal polarisation, the third component (U) is the difference for $\pm 45^\circ$ polarisation and the last component (Q) is the difference between left and right circular polarisation. Further details on polarisation and definition of the Stokes vector are found in Section 22.

ARTS is a fully polarised forward model, but can be run with a smaller number of Stokes components. The selection is made with the workspace variable `stokes_dim`. For example, gaseous absorption and emission are in general unpolarised, and if not scattering has to be considered it is sufficient to only include the first Stokes components in the simulations. To include higher order Stokes components results in this case only in slower calculations. The general case is here denoted as vector radiative transfer, while scalar radiative transfer refers to the case when only the first Stokes component is considered.

3.3 Absorption

Absorption can not yet be calculated internally inside this ARTS version. So far absorption has to be imported, where for example the operational ARTS version (1.0.x) can be used to calculate the needed absorption coefficients. The absorption is put into a look-up table and the absorption is later extracted from this table by interpolation, as described in Section 6.

3.4 Compulsory sensor and data reduction variables

The instrument that detects the simulated radiation is denoted as the sensor. The forward model is constructed in such way that a sensor must exist. For cases when only monochromatic pencil beam radiation is of interest, the positions and directions for which the radiation shall be calculated are given by specifying an imaginary sensor with infinite frequency and angular resolution. The workspace variables for the sensor that always must be specified are `sensor_response`, `sensor_pos`, `sensor_los`, `antenna_dim`, `mblock_za_grid` and `mblock_aa_grid`. These variables are presented separately below. The discussion of sensor workspace variables is continued in Section 10. Section 3.5 gives further insights in how the sensor is treated in ARTS.

3.4.1 Sensor position

The observation positions of the sensor are stored in `sensor_pos`. This is a matrix where each row corresponds to a sensor position. The number of columns in the matrix equals the atmospheric dimensionality (1 column for 1D etc.). The columns of the matrix (from first to last) are radius, latitude and longitude. Accordingly, row i of `sensor_pos` for a 3D case is (r_i, α_i, β_i). The sensor position can be set to any value, but the resulting propagation paths (also dependent on `sensor_los`) must be valid with respect to the model atmosphere (see Section 3.5.2). An obviously incorrect choice is to place the senor below the surface altitude. If the sensor is placed inside the model atmosphere, any sensor line-of-sight is allowed, this including the cases that the sensor is placed on the surface looking down, and that the sensor is placed inside the cloud box.

The fact that the sensor position can be given any value implies that the radius must be used in `sensor_pos`, in contrast to `z_surface` and `z_field` where the altitude above the geoid is applied. This is the case as, for 2D and 3D, the sensor can be placed outside the covered latitude and longitude ranges, thus outside the defined geoid, and the geometrical altitude is undefined.

The sensor is treated to be motionless when calculating the spectrum, or spectra, for each given observation position. One or several spectra can be calculated for each position as described in Section 3.4.4.

3.4.2 Line-of-sight

The viewing direction of the sensor, the line-of-sight, is described by two angles, the zenith angle (ψ) and the azimuth angle (ω). The zenith angle exists for all atmospheric dimensionalities, while the azimuth angle is defined only for 3D. The term line-of-sight is not only used in connection with the sensor, it is also used to describe the local propagation direction along the path taken by the observed radiation (Section 3.5.2). The zenith and azimuthal angles are defined in an identical way in both of these contexts (sensor pointing direction; local propagation direction). This is expected as the position of the sensor is the end point of the propagation path. The sensor line-of-sight is the direction the antenna is pointed to receive the radiation. The line-of-sight for propagation paths is defined likewise, it is the direction in which a hypothetical sensor must be placed to receive the radiation along the propagation path at the point of interest. This means that the line-of-sight and the photons

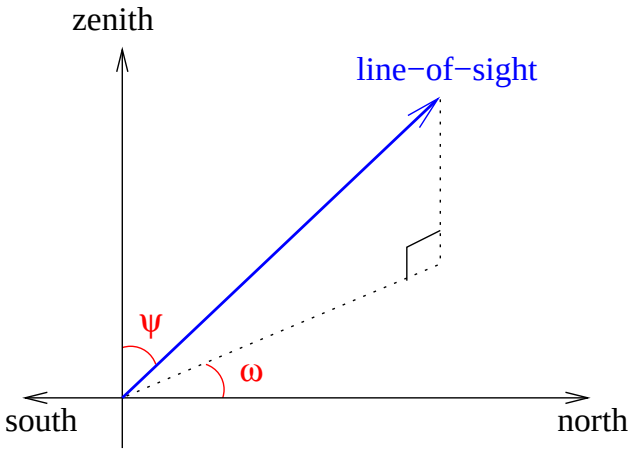

Figure 3.5: Definition of zenith angle, ψ, and azimuth angle, ω, for a line-of-sight. The figure shows a line-of-sight with a negative azimuth angle.

are going in opposite directions. As a true sensor has a finite spatial resolution (described by the antenna pattern), theoretically there is an infinite number of line-of-sights associated with the sensor, but in the forward model, spectra are only calculated for a discrete set of directions. If a sensor line-of-sight is mentioned without any comments, it refers to the direction in which the centre of the antenna pattern is directed.

The zenith angle, ψ, is simply the angle between the line-of-sight and the zenith direction (Figure 3.5). The valid range for 1D and 3D cases is $[0, 180°]$. In the case of 2D, zenith angles down to -180° are also allowed, where the distinction is that positive angles mean a viewing direction towards higher latitudes, and negative angles mean a viewing direction towards lower latitudes. It should be mentioned that the zenith and nadir directions are here defined to be along the line passing the centre of the coordinate system and the point of concern (Section 7.9.1). A nadir observation, $\psi = 180°$, is thus a measurement towards the centre of the coordinate system.

The azimuth angle, ω, is given with respect to the meridian plane. That is, the plane going through the north and south poles of the coordinate system ($\alpha = \pm 90°$) and the sensor. The valid range is $[-180°, 180°]$ where angles are counted clockwise; 0°means that the viewing or propagation direction is north-wise and +90° means that the direction of concern goes eastward. This definition does not work for position on the poles. To cover these special cases, the definition is extended to say that for positions on the poles the azimuth angle equals the longitude along the viewing direction. For example, if standing on any of the poles and the viewing direction is towards Greenwich, the azimuth angle is 0°.

The sensor line-of-sights are stored in `sensor_los`. This workspace variable is a matrix, where the first column holds zenith angles and the second column is azimuth angles. For 1D and 2D there is only one column in the matrix, while for 3D a row i of the matrix is (ψ_i, ω_i). The number of rows for `sensor_los` must be the same as for `sensor_pos`.

3.4.3 Sensor characteristics and data reduction

The term "sensor characteristics" is used here as a comprehensive term for the response of all sensor parts that affect how the field of monochromatic pencil beam intensities are translated to the recorded spectrum. For example, the antenna pattern, the side-band filtering

and response of the spectrometer channels are normally the most important characteristics for a microwave heterodyne radiometer. Any processing of the spectral data that takes place before the retrieval is denoted as data reduction. The most common processing is to represent the original spectra with a smaller set of values, that is, a reduction of the data size. The most common data reduction techniques is binning and Hotelling transformation by an eigenvector expansion.

In ARTS, the influence of sensor characteristics and data reduction is incorporated by transfer matrices, as described in Section 4.2 and 10. The application of these transfer matrices assumes that each step is a linear operation, which should be the case for the response of the parts of a well designed instrument. Non-linear data reduction could be handled by special workspace methods.

The sensor and data reduction are described as a series of units, each having its own transfer matrix. There is only one compulsory transfer matrix and it is `sensor_response`. There are several workspace variables associated with this transfer matrix where `antenna_dim`, `mblock_za_grid` and `mblock_aa_grid` are the compulsory ones.

The variable `antenna_dim` gives the dimensionality of the antenna pattern, where the options are 1 and 2, standing for 1D and 2D, respectively. A 1D antenna dimensionality means that the azimuth extension of the antenna pattern is neglected, there is only a zenith angle variation of the response. A 2D antenna pattern is converted to a 1D pattern by integrating the azimuth response for each zenith angle. For cases with 1D antenna patterns, `mblock_aa_grid` must be set to be an empty vector.

For each sensor position, a number of monochromatic pencil beam spectra are calculated. The monochromatic frequencies are given by `f_grid` (Section 3.3). The pencil beam directions are obtained by summing the sensor line-of-sight angles (`sensor_los`) for the position and the values of `mblock_za_grid` and `mblock_aa_grid`. For example, pencil beam zenith angle i is calculated as

$$\psi_i = \psi_0 + \Delta\psi_i \tag{3.3}$$

where ψ_0 is the sensor line-of-sight for the position of concern and $\Delta\psi_i$ is value i of `mblock_za_grid`. With other words, `mblock_za_grid` and `mblock_aa_grid` give the grid (relative to the sensor line-of-sight) for the calculation of the intensity field that will be weighted with the antenna response.

3.4.4 Measurement sequences and blocks

The series of observations modelled by the simulations is denoted as the measurement sequence. That is, a measurement sequence covers all spectra recorded at all considered sensor positions. A measurement sequence consists of one or several measurement blocks. The observations inside the various blocks differ only with an off-set of the line-of-sight, all other factors should be common for all blocks. A block can be treated as a measurement cycle that is repeated, an integer number of times, to form the measurement sequence. The measurement blocks correspond normally to each unique sensor position of the sequence.

A measurement block covers one or several recorded spectra, depending on the measurement conditions and the atmospheric dimensionality. A block can consist of several spectra when there is no effective motion of the sensor with respect to the atmospheric fields.

It should be noted that for 1D cases, a motion along a constant radius has no influence on the simulated spectra as the same atmospheric fields are seen for a given viewing direction. It is favourable, if possible, to handle all spectra as a single block, instead of using a block for each sensor position. This is the case as the antenna patterns for the different line-of-sights are normally overlapping and a pencil beam spectrum can be used in connection with several measurement spectra to estimate the intensity field. If a measurement sequence is divided into several blocks even if a single block would be sufficient, pencil beam spectra for basically identical propagation paths can be calculated several times, which of course will increase the computational time. To summarise, for cases when the sensor is not in motion, or with a 1D atmosphere and a sensor not moving vertically, the aim should be to use a single block for the measurement sequence.

If not a single block is used, the standard option should be that the blocks cover one spectrum each. There could exist reasons to select an intermediate solution, to let the extent of the blocks be several spectra (but not the full measurement sequence). This could be the case when the atmospheric dimensionality is 2D or 3D, and the sensor is moving but the movement during some subsequent spectra can be neglected. If this can be done must be judged by comparing the movement of the sensor during the extent of the considered block size and the spatial resolution, in the direction of the movement, that is hoped to be achieved. If this intermediate solution shall be an option, the difference in zenith and azimuth angles between the spectra must be the same for all blocks, otherwise `sensor_response` cannot be applied for all blocks as done below in Equation 3.5.

For each block, pencil beam spectra are calculated for the line-of-sights obtained when summing `sensor_los` and `mblock_za_grid` (and possibly `mblock_aa_grid`), as described in Section 3.4.3. The pencil beam spectra for each line-of-sight are appended vertically to form a common vector, $\mathbf{i}_b$. Values are put in following the order in `f_grid`. Hence, the frequencies for this vector are

$$\mathbf{i}_b = \begin{bmatrix} \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_n \end{bmatrix} \\ \vdots \\ \begin{bmatrix} \nu_1 \\ \vdots \\ \nu_n \end{bmatrix} \end{bmatrix} \tag{3.4}$$

where ν_i is element i of `f_grid` and n the length of the same vector. The order of the angles inside `mblock_za_grid` and `mblock_aa_grid` is followed when looping the pencil beam directions, where the azimuth angle direction is the innermost loop. That is, for 2D antenna patterns all azimuth angles are looped for the first zenith angle etc.

The workspace variable `sensor_response` is here denoted as $\mathbf{H}_b$. It is applied on each $\mathbf{i}_b$ and the results are appended vertically, following the order of the positions in `sensor_pos`

$$\mathbf{y} = \begin{bmatrix} \mathbf{H}_b\mathbf{i}_{b,1} \\ \mathbf{H}_b\mathbf{i}_{b,2} \\ \vdots \\ \mathbf{H}_b\mathbf{i}_{b,n} \end{bmatrix} \tag{3.5}$$

where 1 indicates the first sensor position etc. This equation shows that `sensor_response` shall contain at least a description of the antenna response. The matrix `Hb` can also cover other sensor characteristics and data reduction if the features of concern are common for all measurement blocks.

As the sensor line-of-sight and block grid values are just added, there is an ambiguity of the line-of-sight. It is possible to apply a constant off-set to the line-of-sights, if the block grids are corrected accordingly. For example, if the simulations deal with limb sounding and a 1D atmosphere, where normally a single block should be used despite a number of spectra are recorded, it could be practical to set the line-of-sight to the viewing direction of the uppermost or lowermost spectrum, and the zenith angles in `mblock_za_grid` will not be centred around zero which is the case when the "true" line-of-sight is used.

It should be noted that the compulsory sensor variables give no information about the content of the obtained $\mathbf{y}$, as it is not clear which parts and features the block transfer matrix covers. If `Hb` only incorporates the antenna pattern, the result is a set of hypothetical spectra corresponding to a point inside the sensor. On the other hand, if `Hb` includes the whole of the sensor and an eigenvector data reduction, the result is not even a spectrum in traditional way, it is just a column of coefficients with a vague physical meaning.

3.5 Clear sky radiative transfer

An introduction to radiative transfer theory is given in Section 21. This section describes how the radiative transfer equations are solved practically in ARTS. The general presentation assumes that the simulations deal with an emission measurement. Focus is put on emission measurements as ARTS is intended primarily for such observations. However, simulations of transmission measurements are also possible, which is discussed especially in Section 9.2.1.

3.5.1 Calculation procedure

The overall structure of the part solving the radiative transfer equation is fixed. The corresponding workspace method is `RteCalc`. The calculation procedure of `RteCalc` is outlined in Algorithm 1. For further details of each calculation step, see the indicated equation or section.

The primary unit for emission spectra, the unit of $\mathbf{i}_b$ in Equation 3.5, is [W/(Hz·m^2·sr)]. The emission intensity corresponds directly with the definition of the Planck function in Equation 21.46, no scaling terms are applied.

3.5.2 Propagation paths

A pencil beam path through the atmosphere to reach a position along a specific line-of-sight is denoted as the propagation path. Propagation paths are described by a set of points on the path, and the distance along the path between the points. These quantities, and a number of auxiliary variables, are stored together in a structure described in Section 7.3. The path points are primarily placed at the crossings of the path with the atmospheric grids (`p_grid`, `lat_grid` and `lon_grid`). A path point is also placed at the sensor if it is placed inside the atmosphere. Points of surface reflections and tangent points are also included if such

Algorithm 1 Outline of the overall clear sky radiative transfer calculations (`RteCalc`).

```
allocate memory for the matrix y (Equation 3.5)
allocate memory for the matrix i_b (Equation 3.4)
for all sensor positions do
  for all pencil beam directions of the block do (Section 3.4.4)
    determine the propagation path by ppathCalc (Section 3.5.2)
    determine the radiation at the start of the propagation path (Section 3.5.3)
    call rte_agenda
    copy iy to correct part of i_b
  end for
  put the product H_b i_b in correct part of y
end for
```

exist. More points can also be added to the propagation path, for example, by setting an upper limit for the distance along the path between the points.

The propagation paths are determined basically by starting at the sensor and following the path backwards by some ray tracing technique. If the sensor is placed above the model atmosphere, geometrical calculations are used (as there is no refraction in space) to find the crossing between the path and the top of the atmosphere where the ray tracing then starts. Paths are tracked backwards until the top of the atmosphere is reached, or there is an intersection with the cloud box or the surface. The propagation path (or paths) before a surface reflection is calculated when determining the up-welling radiation from the surface (Section 3.5.5). Example on propagation paths are shown in Figures 3.6 and 3.7.

Not all propagation paths are allowed for 2D and 3D. The paths can only enter and leave the model atmosphere at the top of the atmosphere, as the atmospheric fields are treated to be undefined outside the covered latitude and longitude ranges (Figure 3.8). In addition, if the sensor is placed outside the model atmosphere, the line-of-sight zenith angle must be $\geq 90°$, and the tangent point position of the propagation paths must be inside the end points of the latitude and longitude grids, but can be above the top of the atmosphere. Hence, it is allowed that the propagation path is totally outside the atmosphere, as long as the viewing direction is downward and the lowest point of the path, the tangent point, is inside the latitude and longitude limits of the model atmosphere.

Propagation paths are determined by the function `ppathCalc`. This function is normally only called from `RteCalc`, but it can be called separately if needed. The calculation of the path from one crossing of the grids to next crossing is defined by `ppath_step_agenda`. Depending on which function that is selected for `ppath_step_agenda`, refraction will be considered or not, a length criterion between the path points will be applied etc. Functions intended for `ppath_step_agenda` include `ppath_stepGeometric` and `ppath_stepRefractionEuler`.

3.5.3 The radiative background

The radiative intensity at the starting point of the path, and in the direction of the line-of-sight at that point, is denoted as the radiative background. Four possible radiative backgrounds exist:

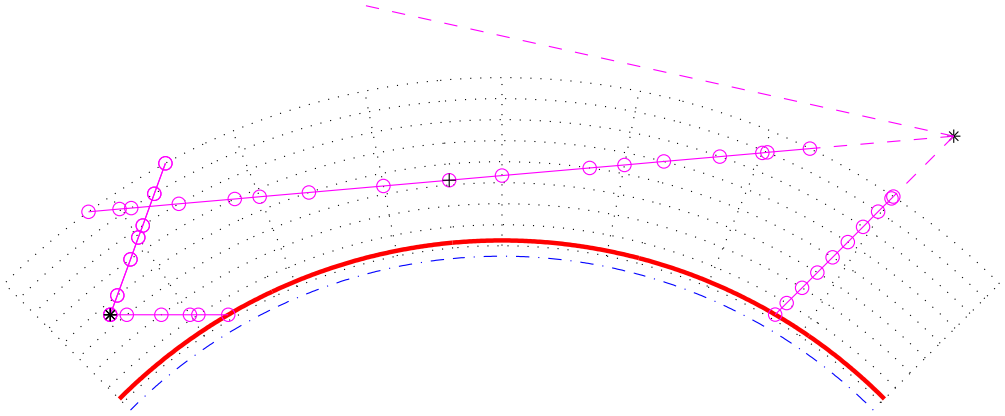

Figure 3.6: Examples on allowed propagation paths for a 2D atmosphere. The atmosphere is plotted as in Figure 3.2 beside that the points for the atmospheric fields are not emphasised. The position of the sensor is indicated by an asterix $(*)$, the points defining the paths are plotted as circles $(\circ)$, joined by a solid line. The part of the path outside the atmosphere, not included in the path structure, is shown by a dashed line. Path points corresponding to a tangent point are marked by an extra plus sign $(\oplus)$. The shown paths include the minimum set of definition points. There exists also the possibility to add points inside the grid cells, for example, to ensure that the distance between the path points does not exceed a specified limit.

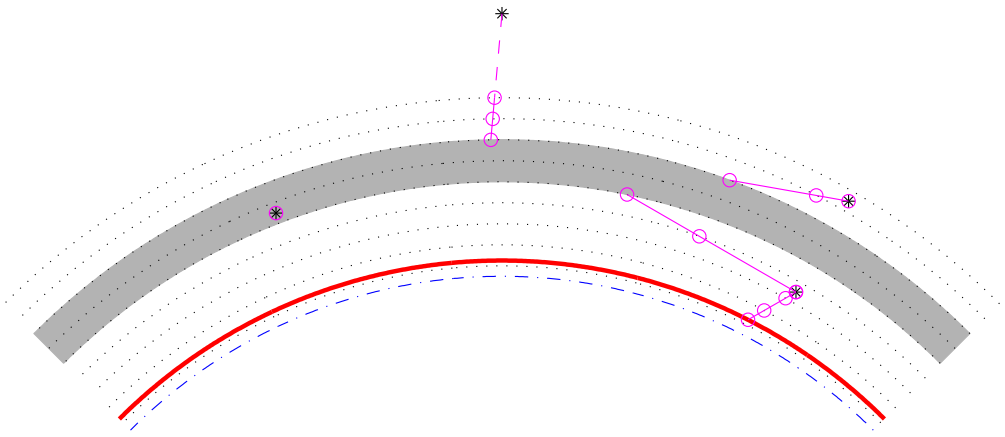

Figure 3.7: Examples on allowed propagation paths for a 1D atmosphere with an activated cloud box. Plotting symbols as in Figure 3.6. When the sensor is placed inside the cloud box, the path is defined with a single point, to know for which position and line-of-sight the intensity field of the cloud box shall be interpolated.

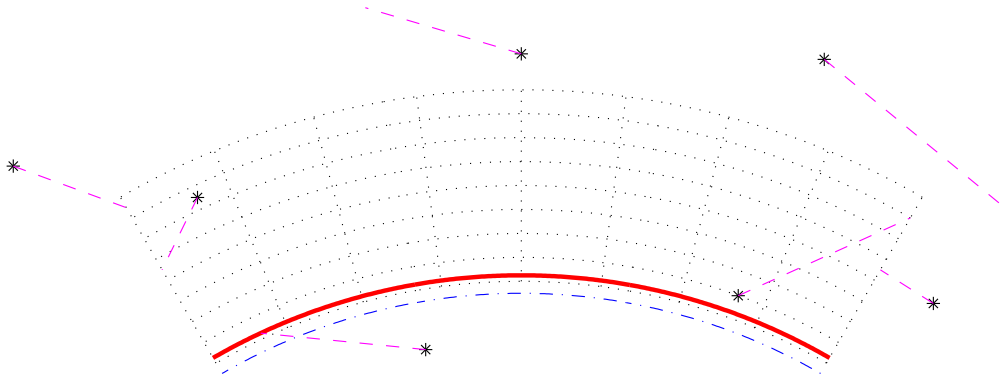

Figure 3.8: Examples on *not* allowed propagation paths for a 2D atmosphere. The constraints for allowed paths are discussed in the text.

Space When the propagation path starts at the top of the atmosphere, space is the radiative background. The normal case should be to set the radiation at the top of the atmosphere to be cosmic background radiation. An exception is when the sensor is directed towards the sun. The radiative background at the top of the atmosphere is determined by `iy_space_agenda`. If a propagation path is totally outside the model atmosphere, the observed monochromatic pencil beam intensity ($\mathbf{i}_b$ in Algorithm 1) equals the output of `iy_space_agenda`.

The surface The sum of surface emission and radiation reflected by the surface is the radiative background when the propagation path intersects with the surface. The calculation of the up-welling radiation from the surface is described in Section 3.5.5.

Surface of cloud box For cases when the propagation path enters the cloud box the radiative background is the intensities leaving the cloud box. This radiation is obtained by `iy_cloudbox_agenda`. How to perform calculations involving scattering are described in Sections 3.6 and 14.

Interior of cloud box If the sensor is situated inside the cloud box, there is basically no propagation path. The radiative background, and also the final spectrum, equals the internal intensity field of the cloud box at the position of the sensor, in the direction of the sensor line-of-sight. This case is also handled by `iy_cloudbox_agenda` and can require some special considerations, as described in Section 3.6.

It should be noted that except for the first case above, the determination of the radiative background involves further radiative transfer calculations. For example, the radiation reflected by the surface can be calculated by a recursive call of `RteCalc` and the radiative background for that calculation is then space or the cloud box. The intensity field entering the cloud box is calculated by calls of `RteCalc` (with cloud box deactivated) and the radiative background is then space or the surface. This results in that space is normally the ultimate radiative background for the calculations. The exception is for propagation paths that intersects with the surface, and the surface is treated to act as a blackbody. For such cases, the propagation path effectively starts at the surface.

3.5.4 The agenda for clear sky radiative transfer, `rte_agenda`

The task of `rte_agenda` is to perform the clear sky radiative transfer calculation along the given propagation path. The determination of the radiative background is not part of the task of `rte_agenda`, it is in the standard case done by `RteCalc` (see Algorithm 1). In fact, methods for `rte_agenda` just assumes that `iy` contains spectral values that make sense and uses these values as start values for the calculations. This means that it is possible to use `rte_agenda` also outside of `RteCalc`, as long as `iy` and `ppath` are set correctly.

The radiative transfer equation can be solved in many ways, and with different level of refinement. The standard approach in ARTS is to solve the radiative transfer from one point of the propagation path to next. The simplest expression, for scalar radiative transfer, that can be used in this way is (cf. Equation 21.54)

$$I_{i+1}(\nu) = I_i(\nu)e^{-\tau_i} + B(\nu, T_i)(1 - e^{-\tau_i}) \tag{3.6}$$

where $I(\nu)$ is the monochromatic (and unpolarised) intensity, i is path step index, τ is the optical thickness along the path of the step, B the Planck function, and T_i is the mean of the temperature at the end points of the step. This expression, and the corresponding ones for vector radiative transfer, are implemented in the method `RteEmissionStd`. All methods for `rte_agenda` adapt automatically to the value of `stokes_dim` (see also Section 3.2).

3.5.5 Surface effects

If there is an interception of the propagation path by the surface, the agenda `iy_surface_agenda` is supposed to provide the upwelling radiation at the interception point along the propagation path. The upwelling radiation is provided by setting `iy`. There exist a standard method for `iy_surface_agenda`, named as `surfaceCalc`. The primary input to this method are the workspace variables `surface_emission`, `surface_los` and `surface_rmatrix`. These three variables define together the properties of the surface, where the expression used to model surface emission and reflections is (Figure 3.9)

$$\mathbf{i}_s^u = \mathbf{i}_e + \sum_l \mathbf{R}_l \mathbf{i}_l^d \tag{3.7}$$

where $\mathbf{i}$ is the Stokes vector for one frequency, $\mathbf{i}_s^u$ is the total upward travelling intensity from the surface along the propagation path, $\mathbf{i}_e$ is the emission from the surface, $\mathbf{i}_l^d$ is the downward travelling intensity reaching the surface along direction l, and $\mathbf{R}_l$ is the reflection coefficient matrix from direction l to the present propagation path. The emission from the surface ($\mathbf{i}_e$) is stored in `surface_emission`, the directions l for which downward travelling intensities are given by `surface_los`, and the reflection coefficients ($\mathbf{R}$) are stored in `surface_rmatrix`. Surface reflections and emission are discussed further in Section 8.

3.5.6 Calculation accuracy

The accuracy of the calculations depends on many factors. For many factors, such as spectroscopic parameters, there is nothing else to do than using best avaliable data. On the

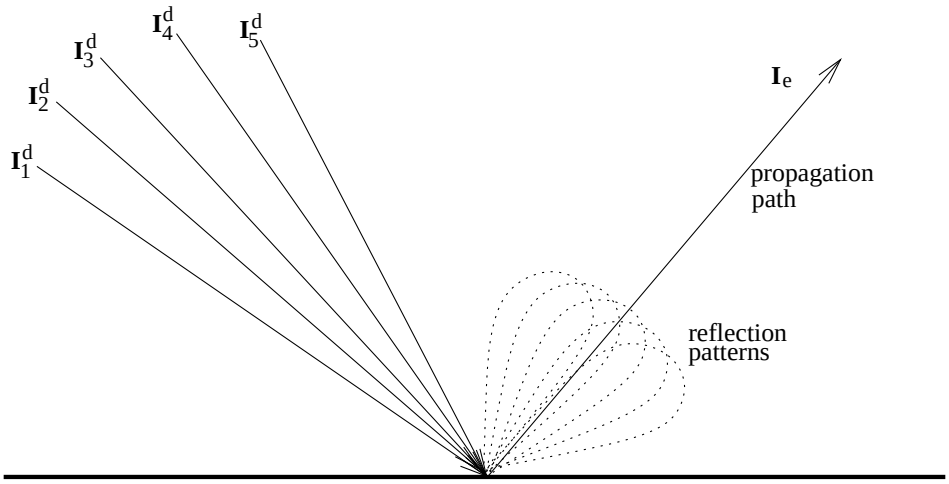

Figure 3.9: Schematic of Equation 3.7.

other hand, for other factors there is a trade-off between accuracy and speed. More accurate calculations requires normally also more computer memory. All different grids and the propagation path step length fall into this category of accuracy factors. It could be worth discussing the selection of atmospheric grids and the path step length as there can be some confusion about how that affects the accuracy.

The main purpose of the atmospheric grids (`p_grid`, `lat_grid` and `lon_grid`) is to build up the mesh on which the atmospheric fields are defined. This means that the spacing of these grids shall be selected having the representation of the atmospheric fields in mind. That is, the spacing shall be fine enough that the atmospheric field is sufficiently well approximated by the piecewise (multi-)linear representation between the grid crossings. The result is that a finer spacing must be used to represent correctly atmospheric fields with a lot of structure, while the grids can have fewer points when the atmospheric fields are smooth.

The accuracy when performing the actual radiative transfer calculations depends on the refinement of the expressions used and the discretisation of the propagation path. If Equation 3.6 is used, the underlaying assumptions are that the temperature is constant, and that the absorption varies linearly, along the propagation path step. These assumptions are of course less violated if the path step length is made small. An upper limit of the path step length is set by the generic input argument `lmax`. In many cases it should suffice to just include path points at the crossings of the atmospheric grids (`lmax`≤ 0). An exception can be limb sounding where the path step length can be very long around the tangent point, but a limit of about 25 km should suffice normally.

As points are always included in the propagation paths at the crossings of the atmospheric grids, finer grids will give shorter path steps. However, it is neither good practice or efficient to use the atmospheric grids to control the accuracy of the radiative transfer

calculations. An upper limit on the path step length shall be applied for this purpose.[1]

3.6 Scattering

The scattering inside the cloud box can be handled by several methods. The different modules are here only quickly outlined, details are given in seperate chapters.

3.6.1 DOIT – the discrete ordinate iterative module

In the DOIT module, the complete intensity field inside the cloud box is determined. This means that the intensity from a discrete set of directions is calculated at all positions of the grid mesh inside the cloud box. The intensity field is determined in an iterative manner, and this calculation must be performed before calling `RteCalc`. The DOIT module is described in Chapter 13.

3.6.2 MC – reversed Monte Carlo scattering module

The approach in the MC module is to follow propagation paths backwards, with scattering angles and path lengths randomly chosen from probability density functions determined by the scattering phase function, and a scalar extinction coefficient, respectively. The phase matrices for every scattering event and scalar extinction are then sequentially applied to the source Stokes vector to give the Stokes vector contribution for each photon. The tracking of the propagation paths starts at the cloud box boundary and continues backwards to the point of emission or the boundary of the cloud box (whatever comes first). See further Chapter 14.

[1] Further discussion can be found in message 399 and 410 of the ARTS developers mailing list.

Part II

Algorithm Descriptions

Chapter 4

Theoretical formalism

In this section, a theoretical framework of the forward model is presented. The presentation follows *Rodgers* [1990], but some extensions are made, for example, the distinction between the atmospheric and sensor parts of the forward model is also discussed. After this chapter was written, C.D. Rodgers published a textbook [*Rodgers*, 2000] presenting the formalism in more detail than *Rodgers* [1990]. Modelling of sensor characteristics is not yet included in ARTS (this part is so far covered by AMI), but treatment of the sensor is here included for completeness.

4.1 The forward model

The radiative intensity, I, at a point in the atmosphere, r, for frequency ν and traversing in the direction, ψ, depends on a variety of physical processes and continuous variables such as the temperature profile, T:

$$I = F(\nu, r, \psi, T, \ldots) \tag{4.1}$$

To detect the spectral radiation some kind of sensor, having a finite spatial and frequency resolution, is needed, and the observed spectrum becomes a vector, $\mathbf{y}$, instead of a continuous function. The atmospheric radiative transfer is simulated by a computer model using a limited number of parameters as input (that is, a discrete model), and the forward model, $\mathcal{F}$, used in practice can be expressed as

$$\mathbf{y} = \mathcal{F}(\mathbf{x}_\mathcal{F}, \mathbf{b}_\mathcal{F}) + \varepsilon(\mathbf{x}_\varepsilon, \mathbf{b}\varepsilon) \tag{4.2}$$

where $\mathbf{x}_\mathcal{F}$, $\mathbf{b}_\mathcal{F}$, $\mathbf{x}_\varepsilon$ and $\mathbf{b}_\varepsilon$ together give a total description of both the atmospheric and sensor states, and ε is the measurement errors. The parameters are divided in such way that $\mathbf{x}$, the state vector, contains the parameters to be retrieved, and the remainder is given by $\mathbf{b}$, the model parameter vector. The total state vector is

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_\mathcal{F} \\ \mathbf{x}_\varepsilon \end{bmatrix} \tag{4.3}$$

History

000306 Written by Patrick Eriksson, partly based on *Eriksson* [1999] and *Eriksson et al.* [2000].

and the total model parameter vector is

$$\mathbf{b} = \begin{bmatrix} \mathbf{b}_{\mathcal{F}} \\ \mathbf{b}_{\varepsilon} \end{bmatrix} \tag{4.4}$$

The actual forward model consists of either empirically determined relationships, or numerical counterparts of the physical relationships needed to describe the radiative transfer and sensor effects. The forward model described here is mainly of the latter type, but some parts are more based on empirical investigations, such as the parameterisations of continuum absorption.

Both for the theoretical formalism and the practical implementation, it is suitable to make a separation of the forward model into two main sections, a first part describing the atmospheric radiative transfer for pencil beam (infinite spatial resolution) monochromatic (infinite frequency resolution) signals [*Eriksson*, 1999],

$$\mathbf{i} = \mathcal{F}_r(\mathbf{x}_r, \mathbf{b}_r) \tag{4.5}$$

and a second part modelling sensor characteristics,

$$\mathbf{y} = \mathcal{F}_s(\mathbf{i}, \mathbf{x}_s, \mathbf{b}_s) + \varepsilon(\mathbf{x}_\varepsilon, \mathbf{b}_\varepsilon) \tag{4.6}$$

where $\mathbf{i}$ is the vector holding the spectral values for the considered set of frequencies and viewing angles ($\mathbf{i}^i = I(\nu^i, \psi^i, \ldots)$, where i is the vector index), and $\mathbf{x}_{\mathcal{F}}$ and $\mathbf{b}_{\mathcal{F}}$ are separated correspondingly, that is, $\mathbf{x}_{\mathcal{F}}^T = [\mathbf{x}_r^T, \mathbf{x}_s^T]$ and $\mathbf{b}_{\mathcal{F}}^T = [\mathbf{b}_r^T, \mathbf{b}_s^T]$. The vectors $\mathbf{x}$ and $\mathbf{b}$ can now be expressed as

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_r \\ \mathbf{x}_s \\ \mathbf{x}_\varepsilon \end{bmatrix} \tag{4.7}$$

and

$$\mathbf{b} = \begin{bmatrix} \mathbf{b}_r \\ \mathbf{b}_s \\ \mathbf{b}_\varepsilon \end{bmatrix}, \tag{4.8}$$

respectively. The subscripts of $\mathbf{x}$ and $\mathbf{b}$ are below omitted as the distinction should be clear by the context.

4.2 The sensor transfer matrix

The modelling of the different sensor parts can be described by a number of of analytical expressions (see *Eriksson and Merino* [1997]) that together makes the basis for the sensor model. These expressions are throughout linear operations and it possible, as suggested in *Eriksson et al.* [2000], to implement the sensor model as a straightforward matrix multiplication:

$$\mathbf{y} = \mathbf{H}\mathbf{i} + \varepsilon \tag{4.9}$$

where $\mathbf{H}$ is here denoted as the sensor transfer matrix. The matrix $\mathbf{H}$ can further incorporate effects of a data reduction and the total transfer matrix is then

$$\mathbf{H} = \mathbf{H}_d\mathbf{H}_s \tag{4.10}$$

as

$$\mathbf{y} = \mathbf{H}_d\mathbf{y}' = \mathbf{H}_d(\mathbf{H}_s\mathbf{i} + \varepsilon') = \mathbf{H}\mathbf{i} + \varepsilon \tag{4.11}$$

where $\mathbf{H}_d$ is the data reduction matrix, $\mathbf{H}_s$ the sensor matrix, and $\mathbf{y}'$ and ε' are the measurement vector and the measurement errors, respectively, before data reduction.

4.3 Weighting functions

4.3.1 Basics

A weighting function is the partial derivative of the spectrum vector $\mathbf{y}$ with respect to some variable used by the forward model. As the input of the forward model is divided between $\mathbf{x}$ or $\mathbf{b}$, the weighting functions are divided correspondingly between two matrices, the state weighting function matrix

$$\mathbf{K}_\mathbf{x} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \tag{4.12}$$

and the model parameter weighting function matrix

$$\mathbf{K}_\mathbf{b} = \frac{\partial \mathbf{y}}{\partial \mathbf{b}} \tag{4.13}$$

For the practical calculations of the weighting functions, it is important to note that the atmospheric and sensor parts can be separated. For example, if $\mathbf{x}$ only hold atmospheric and spectroscopic variables, $\mathbf{K}_\mathbf{x}$ can be expressed as

$$\mathbf{K}_\mathbf{x} = \frac{\partial \mathbf{y}}{\partial \mathbf{i}}\frac{\partial \mathbf{i}}{\partial \mathbf{x}} = \mathbf{H}\frac{\partial \mathbf{i}}{\partial \mathbf{x}} \tag{4.14}$$

This equation shows that the new parts needed to calculate atmospheric weighting functions, are functions giving $\partial\mathbf{i}/\partial\mathbf{x}$ where $\mathbf{x}$ can represent the vertical profile of a species, atmospheric temperatures, spectroscopic data etc.

4.3.2 Transformation between vector spaces

It could be of interest to transform a weighting function matrix from one vector space to another[1]. The new vector, $\mathbf{x}'$, is here assumed to be of length n ($\mathbf{x}' \in \mathbf{R}^{n\times 1}$), while the original vector, $\mathbf{x}$ is of length p ($\mathbf{x} \in \mathbf{R}^{p\times 1}$). The relationship between the two vector spaces is described by a transformation matrix $\mathbf{B}$:

$$\mathbf{x} = \mathbf{B}\mathbf{x}' \tag{4.15}$$

where $\mathbf{B}\in\mathbf{R}^{p\times n}$. For example, if $\mathbf{x}'$ is assumed to be piecewise linear, then the columns of $\mathbf{B}$ contain tenth functions, that is, a function that are 1 at the point of interest and decreases

[1] This subject is also discussed in *Rodgers* [2000], published after writing this.

linearly down to zero at the neighbouring points. The matrix can also hold a reduced set of eigenvectors.

The weighting function matrix corresponding to $\mathbf{x}'$ is

$$\mathbf{K}_{\mathbf{x}'} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}'} \tag{4.16}$$

This matrix is related to the weighting function matrix of $\mathbf{x}$ (Eq. 4.12) as

$$\mathbf{K}_{\mathbf{x}'} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{x}'} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \mathbf{B} = \mathbf{K}_{\mathbf{x}} \mathbf{B} \tag{4.17}$$

Note that

$$\mathbf{K}_{\mathbf{x}'} \mathbf{x}' = \mathbf{K}_{\mathbf{x}} \mathbf{B} \mathbf{x}' = \mathbf{K}_{\mathbf{x}} \mathbf{x} \tag{4.18}$$

However, it should be noted that this relationship only holds for those $\mathbf{x}$ that can be represented perfectly by some $\mathbf{x}'$ (or vice versa), that is, $\mathbf{x} = \mathbf{B}\mathbf{x}'$, and not for all combinations of $\mathbf{x}$ and $\mathbf{x}'$.

If $\mathbf{x}'$ is the vector to be retrieved, we have that [*Rodgers*, 1990]

$$\hat{\mathbf{x}}' = \mathcal{I}(\mathbf{y}, \mathbf{c}) = \mathcal{T}(\mathbf{x}, \mathbf{b}, \mathbf{c}) \tag{4.19}$$

where $\mathcal{I}$ and $\mathcal{T}$ are the inverse and transfer model, respectively.

The contribution function matrix is accordingly

$$\mathbf{D}_{\mathbf{y}} = \frac{\partial \hat{\mathbf{x}}'}{\partial \mathbf{y}} \tag{4.20}$$

that is, $\mathbf{D}_{\mathbf{y}}$ corresponds to $\mathbf{K}_{\mathbf{x}'}$, not $\mathbf{K}_{\mathbf{x}}$.

We have now two possible averaging kernel matrices

$$\mathbf{A}_{\mathbf{x}} = \frac{\partial \hat{\mathbf{x}}'}{\partial \mathbf{x}} = \frac{\partial \hat{\mathbf{x}}'}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \mathbf{D}_{\mathbf{y}} \mathbf{K}_{\mathbf{x}} \tag{4.21}$$

$$\mathbf{A}_{\mathbf{x}'} = \frac{\partial \hat{\mathbf{x}}'}{\partial \mathbf{x}'} = \frac{\partial \hat{\mathbf{x}}'}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{x}'} = \mathbf{D}_{\mathbf{y}} \mathbf{K}_{\mathbf{x}'} = \mathbf{A}_{\mathbf{x}} \mathbf{B} \tag{4.22}$$

where $\mathbf{A}_{\mathbf{x}} \in \mathbf{R}^{p \times n}$ and $\mathbf{A}_{\mathbf{x}'} \in \mathbf{R}^{p \times p}$, that is, only $\mathbf{A}_{\mathbf{x}'}$ is square. If $p > n$, $\mathbf{A}_{\mathbf{x}}$ gives more detailed information about the shape of the averaging kernels than the standard matrix ($\mathbf{A}_{\mathbf{x}'}$). If the retrieval grid used is coarse, it could be the case that $\mathbf{A}_{\mathbf{x}'}$ will not resolve all the oscillations of the averaging kernels, as shown in *Eriksson* [1999, Figure 11].

Chapter 5

Description of the atmosphere

FIXME: Patrick: Move parts from chapter 3.1 here

5.1 Atmospheric fields

5.1.1 Gridded Fields

In order to store three-dimensional atmospheric fields along with the atmospheric grids, the class `GField3` was implemented. It resides in the files `gridded_fields.h` and `gridded_fields.cc`. The reading routine `AtmRawRead` requires the volume mixing ratio profiles of all gas species and the altitude and temperature profiles in `GField3` format. The reading routines `ParticleTypeAdd` and `ParticleTypeAddAll` require the particle number density fields in `GField3` format.

The `GField3` consists of the following fields:

- *Vector* `p_grid`: Pressure grid [Unit: Pa].
- *Vector* `lat_grid`: Latitude grid [Unit: °].
- *Vector* `lon_grid`: Longitude grid [Unit: °].
- *Tensor3* `data`: Data of the atmospheric field. The dimensions of the *Tensor3* are: `[pressure latitude longitude]`
  The unit is chosen according to the atmospheric field.

History

050913 Created by Claudia Emde. Included GField3 description.

Chapter 6

Gas absorption

6.1 The gas absorption lookup table

6.1.1 Introduction

Calculating gas absorption coefficient spectra in a line by line way is quite an expensive thing to do. Sometimes contributions from thousands or ten thousands of lines have to be summed up. To make matters worse, this has to be done over and over again for each point in the atmosphere.

Actually, the absorption coefficient depends not directly on position, but on the atmospheric state variables:

- Pressure
- Temperature
- Trace gas concentrations

The basic idea of the lookup table is to pre-calculate absorption for discrete combinations of these variables, and then use interpolation to extract absorption for the actual atmospheric state.

History	
2002-06-04	Restarted for ARTS-1-1 by Stefan Buehler.
2003-03-10	Lookup tables added by Stefan Buehler.
2003-03-28	Documentation for WSM abs_fieldCalc extended by Stefan Buehler after comment from Sreerekha T. R..

Here	Unit	In ARTS	Description
I	$\frac{\mathrm{W}}{\mathrm{m}^2\ \mathrm{Hz}\ \mathrm{sr}}$	i_rte, i_field, ...	Intensity
l	m		Path length element
κ_i	m^2	xsec	Absorption cross section of absorbing species i
n_i	m^{-3}		Number density of species i
α_i	m^{-1}	abs_scalar_gas	Absorption coefficient of absorbing species i
α_{total}	m^{-1}		Total gas absorption coefficient

Table 6.1: Examples of symbols used in this chapter, the corresponding notation in the ARTS source code and a short description of the quantity.

6.1.2 Lookup table concept

The fundamental law of Beer[1] states that extinction is proportional to the intensity of radiation, and to the amount of absorbing substance:

$$\frac{dI}{dl} = -I \sum_i \kappa_i n_i = -I \sum_i \alpha_i = -I \alpha_{\text{total}} \tag{6.1}$$

where the meaning of the symbols is defined in Table 6.

As one can see from the above equation, a large part of the pressure dependence of α_i comes from n_i. (If one assumes constant volume mixing ratio of species i, then n_i is proportional to the total pressure according to the ideal gas law.) Therefore, the lookup table should store κ, rather than α. We then have to worry only about the dependence of κ on the atmospheric state variables.

Pressure dependence

The pressure dependence is the most important dependence of κ. It comes from the fact that the width of the line shape functions is governed by pressure broadening. We have to store the κ_i on some pressure grid and interpolate if we need them for intermediate values.

Temperature dependence

This is the next effect to take into account. Both the line widths and the line intensities depend on temperature. Of course, only certain combinations of pressure and temperature occur in the Earth's atmosphere. Hence, storing the κ_i in a two dimensional table as a function of pressure and temperature would waste a lot of space. Instead, they are stored for a reference temperature and set of temperature perturbations for each pressure level. E.g., if the set of perturbations is $[-10,\ 0,\ +10]$, then the κ_i would be stored for three different temperatures for each pressure level: $[T_R(p) - 10\,\mathrm{K},\ T_R(p),\ T_R(p) + 10\,\mathrm{K}]$, where $T_R(p)$ is the reference temperature for each pressure level.

[1] According to C. Melsheimer, Beer's law is: 'The taller the glass, the darker the brew, the less the amount of light that comes through'. He might have been quoting someone else, there, but I do not know whom.

Trace gas concentration dependence

This is a second order effect. The width of the line depends not only on total pressure, but also on the partial pressure of one or more trace gases. In theory this is always the case, because the broadening is different for each combination of collision partners. However, in practice trace gas concentrations in the Earth's atmosphere are normally so low that this can be safely neglected. An important exception is water vapor in the lower troposphere, which can reach quite high volume mixing ratios. Therefore, the effect of water vapor mixing ratio on water vapor absorption (self broadening), as well as on oxygen absorption (according to a parameterization by *Rosenkranz* [1993]) may not be negligible.

This is handled by storing perturbations, similar to the temperature case. The user can select for which species perturbations should be stored. (The so called 'nonlinear species'.)

This feature is not yet used. FIXME: Update this when it works.

6.1.3 Implementation

The gas absorption lookup table is implemented by the class `GasAbsLookup`, which resides in the files `gas_abs_lookup.cc` and `gas_abs_lookup.h`.

Lookup table structure

Below you find the actual declaration of the class GasAbsLookup with extensive comments.

```
//! An absorption lookup table.
/*! This class holds an absorption lookup table, as well as all
    information that is necessary to use the table to extract
    absorption. Extraction routines are implemented as member
    functions. */
struct GasAbsLookup {
public:
  // Documentation is with the implementation!
  void Adapt( const ArrayOfArrayOfSpeciesTag& current_species,
              ConstVectorView current_f_grid );

  // Documentation is with the implementation!
  void Extract( Matrix&         sga,
                              const Index&    f_index,
                              const Numeric&  p,
                              const Numeric&  T,
                ConstVectorView abs\_vmrs ) const;

    // Obsolete try for a function to extract for the entire field:
    //   void Extract( Tensor5View      sga,
    //                 const Index&     f_index,
    //                 ConstVectorView  p,
    //                 ConstTensor3View T,
    //                 ConstTensor4View abs\_vmrs ) const;

  // IO functions must be friends:
  friend void xml_read_from_stream( istream& is_xml,
                                    GasAbsLookup& gal,
                                    bifstream *pbifs );
  friend void xml_write_to_stream ( ostream& os_xml,
```

```
                                    const GasAbsLookup& gal,
                                    bofstream *pbofs );

private:

  //! The species tags for which the table is valid.
  ArrayOfArrayOfSpeciesTag species;

  //! The species tags with non-linear treatment.
  /*! This must be inside the range of species. If nonlinear_species
    is an empty vector, it means that all species should be treated
    linearly. (No absorption for perturbed species profiles is
    stored.) */
  ArrayOfIndex nonlinear_species;

  //! The frequency grid [Hz].
  /*! Must be sorted in ascending order. */
  Vector    f_grid;

  //! The pressure grid for the table [Pa].
  /*! Must be sorted in decreasing order. */
  Vector    p_grid;

  //! The reference VMR profiles.
  /*! The VMRs for all species, associated with p_grid. Dimension:
    [n_species, n_p_grid]. These VMRs are needed to scale the
    absorption coefficient to other VMRs. We are never working with
    "absorption cross-sections", always with real absorption
    coefficients, so we have to remember the associated VMR values.

    Physical unit: Absolute value. */
  Matrix    vmrs_ref;

  //! The reference temperature profile [K].
  /*! This is a temperature profile. The dimension must be the same as
    p_grid. */
  Vector    t_ref;

  //! The vector of temperature perturbations [K].
  /*! This can have any number of elements. Example:
    [-20,-10,0,10,20]. The actual temperatures for which absorption is
    stored are t_ref + t_pert for each level. The reference
    temperature itself should normally also be included, hence
    t_pert should always include 0. Must be sorted in ascending order!

    The vector t_pert may be an empty vector (nelem()=0), which marks
    the special case that no interpolation in temperature should be
    done. If t_pert is not empty, you will get an error message if you
    try to extract absorption for temperatures outside the range of
    t_pert. */
  Vector    t_pert;

  //! The vector of perturbations for the VMRs of the nonlinear species.
  /*!
    These apply to all the species that have been set as
```

```
    nonlinear_species.

    Fractional units are used! Example: [0,.5,1,10,100],
    meaning from VMR 0 to 100 times the profile given in
    abs\_vmrs. The reference value should normally be included, hence
    nls_pert should always include the value 1.

    If nonlinear_species is an empty vector, it means that there are
    no nonlinear species. Then nls_pert must also be an empty vector.
  */
  Vector    nls_pert;

  //! Absorption cross sections.
  /*!
    Physical unit: m^2

    \attention We want to interpolate these beasts in pressure. To
    keep interpolation errors small it is better to store
    cross-sections, not coefficients. The absorption coefficient alpha
    is given by alpha = xsec * n, where n is the number density.

    Dimension: [ a, b, c, d ]

    Simplest case (no temperature perturbations,
    no vmr perturbations): <br>
    a = 1 <br>
    b = n_species <br>
    c = n_f_grid <br>
    d = n_p_grid <br>

    Standard case (temperature perturbations,
    but no vmr perturbations): <br>
    a = n_t_pert <br>
    b = n_species <br>
    c = n_f_grid <br>
    d = n_p_grid <br>

    Full case (with temperature perturbations
    and vmr perturbations): <br>
    a = n_t_pert <br>
    b = n_species + n_nonlinear_species * ( n_nls_pert - 1 ) <br>
    c = n_f_grid <br>
    d = n_p_grid <br>

    Note that the last three dimensions are identical to the
    dimensions of abs_per_tg in ARTS-1-0. This should simplify
    computation of the lookup table with this old ARTS version.
  */
  Tensor4 xsec;
};
```

Workspace variables and methods

The lookup table itself is stored in the WSV `abs_lookup`. After loading (with `ReadXML`), it is important that one calls the WSM `abs_lookupAdapt`. This will make sure that

the lookup table agrees exactly with your calculation. For example, it has to check that the frequencies that you want to use are included in the set of frequencies for which the table has been calculated. **There is no interpolation in frequency!** This is on purpose, because the gas absorption spectrum is the quantity that changes most rapidly as a function of frequency. Frequency interpolation here would be stupid and dangerous. The method also sorts the species in exactly the same way that they occur in your calculation. It sets the WSV `abs_lookup_is_adapted` to flag that the table is now ok.

When the table has been successfully adapted, one can extract absorption coefficients with the WSM `abs_scalar_gasExtractFromLookup`. This will extract *absorption coefficients*, i.e., the cross sections stored in the table are not only interpolated to the desired atmospheric conditions, but are also multiplied with the partial number density of the present absorbers.

The `abs_scalar_gasExtractFromLookup` method is meant to be used inside the agenda `abs_scalar_gas_agenda`, which is used in several places where absorption coefficients are needed, both inside the scattering box and outside.

It is also possible to calculate absorption for the entire atmospheric field. This is done by the method `abs_fieldCalc`, which is useful in two different contexts:

1. For testing and plotting gas absorption. (For RT calculations, gas absorption is calculated or extracted locally, therefore there is no need to calculate a global field. But this method is handy for easy plotting of absorption vs. pressure, for example.)

2. Inside the scattering region, monochromatic absorption is pre-calculated for the entire atmospheric field. **FIXME: At least that's the plan, isn't it Claudia? Please remove this FIXME when that works.**

Because of the different usage contexts, the method `abs_fieldCalc` can calculate absorption either for all frequencies in the frequency grid (input variable `f_index`<0), or just for the frequency indicated by the input variable `f_index` (`f_index`$>=0$).

The following controlfile section illustrates the use of the lookup table together with `abs_fieldCalc`. This is not a complete controlfile. **FIXME: Eventually there should be a good complete example in doc/examples. For the moment you can look at the examples in sbuehler/arts_calc/abs_lookup_2/.**

```
# Read lookup table:
ReadXML( abs_lookup, "some_table.xml" )

# Adapt lookup table:
abs_lookupAdapt

# Set agenda for extracting absorption
AgendaSet( abs_scalar_gas_agenda ){
  abs_scalar_gasExtractFromLookup
}

# Input to abs_scalar_gasExtractFromLookup{},
# means to calculate all frequencies.
```

```
IndexSet( f_index, -1 )

# Calculate scalar gas absorption field. (This assumes that
# also the WSVs f_grid, atmosphere_dim, p_grid, lat_grid,
# lon_grid, t_field, and vmr_field have been defined.)
abs_fieldCalc

# Write out the field:
WriteXML( "ascii", abs_field, "" )
```

Use the online documentation for the methods and variables mentioned to learn more.

Chapter 7

Propagation paths and the geoid

A propagation path is the way the radiation travels to reach the sensor for a specified line-of-sight. A general description of propagation paths is given in Section 3.5.2 and it can be a good idea to read that section before continuing here. This section describes how propagation paths are described and calculated. In addition, at the end of the section some geodetic issues are discussed, such as the choice of reference ellipsoid for the geoid.

7.1 Implementation files

Variables and functions related to propagation paths are defined in the files:

- `ppath.h`
- `ppath.cc`
- `m_ppath.cc`
- `m_atmosphere.cc`

The first file, `ppath.h`, contains the definition of the structure to describe propagation paths, `Ppath`. The second file, `ppath.cc`, contains functions to perform calculations to determine propagation paths. The third file, `m_ppath.cc`, contains the workspace methods related to propagation paths, but these methods mainly check the input and the actual calculations are performed by sub-functions in `ppath.cc`. The fourth file, `m_atmosphere.cc`, contains methods to set the geoid radius.

7.2 Calculation approach

The propagation paths are calculated in steps, as outlined below in this section. The path steps are normally from one crossing of the atmospheric grids to next. This solution is

History	
050613	Some new features described by Patrick Eriksson.
030310	First complete version written by Patrick Eriksson.

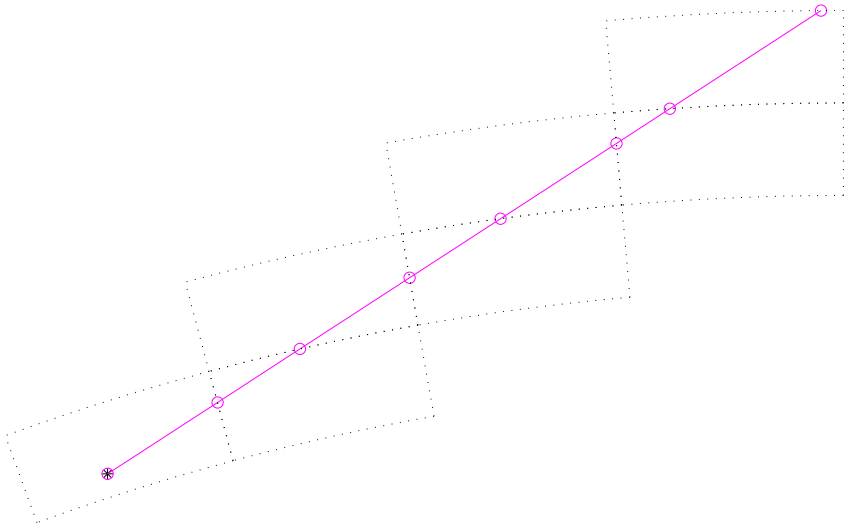

Figure 7.1: Tracking of propagation paths. For legend, see Figure 7.2. The figure tries to visualize how the calculations of propagation paths are performed from one grid cell to next. In this example, the calculations start directly at the sensor position $(*)$ as it placed inside the model atmosphere. The circles give the points defining the propagation path. Path points are always included at the crossings of the grid cell boundaries. Such a point is then used as the starting point for the calculations inside the next grid cell.

necessary to allow that the same code is used throughout the program. To introduce propagation paths steps was necessary to handle the iterative solution for scattering inside the cloud box, as made clear from Figure 13.2.

A full propagation path is stored in the workspace variable `ppath`, that is of the type `Ppath` (see Section 7.3). The paths are determined by calculating a number of path steps. A path step is the path from a point to the next crossing of either the pressure, latitude or longitude grid (Figure 7.1). There is one exception to this definition of a path step, and that is when there is an intersection with the surface, which ends the propagation path at that point. The starting point for the calculation of a path step is normally a grid crossing point, but can also be an arbitrary point inside the atmosphere, such as the sensor position. Only points inside the model atmosphere are handled. The path steps are stored in the workspace variable `ppath_step`, that is of the same type as `ppath`. The path steps are calculated by an agenda called `ppath_step_agenda`. Example on methods that can be used in `ppath_step_agenda` are `ppath_stepGeometric` and `ppath_stepRefractionEuler`.

Propagation paths are calculated with the workspace method `ppathCalc`. The communication between this method and `ppath_step_agenda` is handled by `ppath_step`. That variable is used both as input and output to `ppath_step_agenda`. The agenda gets back `ppath_step` as returned to `ppathCalc` and the last path point hold by the structure is accordingly the starting point for the new calculations. If a total propagation path shall be determined, the agenda is called repeatedly until the starting point of the propagation path is found and `ppath_step` will hold all path steps that together make up `ppath`. The

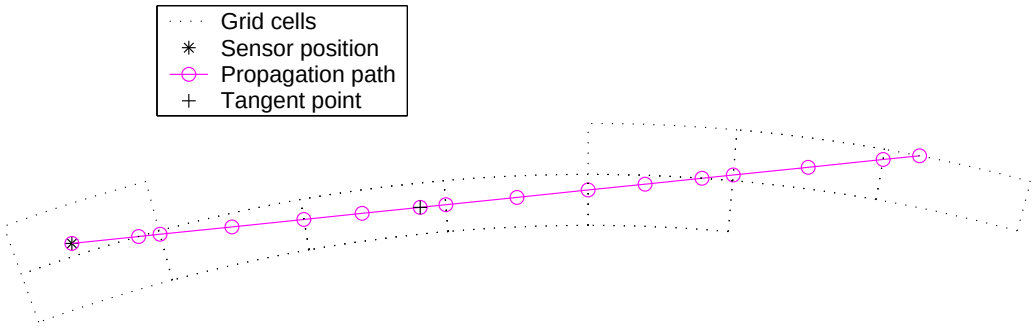

Figure 7.2: As Figure 7.1, but with a length criterion for the distance between the points defining the path. The inclusion of the tangent point is not a result of this length criterion, it is always included among the path points.

starting point is included in the returned structure.

The path is determined by starting at the end point and moving backwards to the starting point. The calculations are initiated by filling `ppath_step` with the practical end point of the path. This is either the position of the sensor (true or hypothetical), or some point at the top of the atmosphere (determined by geometrical calculations starting at the sensor). This initialization is not handled by `ppath_step_agenda`. The field `constant` is set by `ppathCalc` to the correct value if the sensor is above the model atmosphere. Otherwise, the field is set to be negative and is corrected by `ppath_step_agenda` at the first call. This procedure is needed as the propagation path constant changes if refraction is considered, or not, when the sensor is placed inside the atmosphere.

The agenda performs only calculations to next crossing of a grid, all other tasks are performed by `ppathCalc`, with one exception. If there is an intersection with the surface, the calculations stop at this point. This is flagged by setting the background field of `ppath_step`. Beside this, `ppathCalc` checks if the starting point of the calculations is inside the scattering box or below the surface level, and check if the last point of the path has been reached.

The `ppath_step_agenda` put in points along the propagation path at all crossings with the grids, tangent points and points of surface reflection. Additional points can be included in the propagation paths. For example, an upper distance between the points defining the path can be set for `ppath_stepGeometric` by the generic input `lmax` (see Figure 7.2).

In many cases the propagation path can/must be considered to consist of several parts. One exemple is surface reflection (see Figure 3.9). The variable `ppath` describes then only a single part of the propagation path, while the complete path is put into `ppath_array`. The part closest to the sensor is first in this array (index 0). The order of the following parts is described by the `next_parts` field of the `Ppath` structure (see below).

7.3 The propagation path data structure

A propagation path is represented by a structure of type `Ppath`. This structure holds also auxiliary variables to facilitate the radiative transfer calculations and to speed up the interpolation. The fields of `Ppath` are described below, where the data type is given inside square brackets.

dim [Index] The atmospheric dimensionality. This field shall always be equal to the workspace variable `atmosphere_dim`.

np [Index] Number of positions to define the propagation path. Allowed values are ≥ 0. The number of rows of `pos` and `los`, and the length of `z`, `gp_p`, `gp_lat` and `gp_lon`, shall be equal to `np`. The length of `l_step` is `np` - 1. If `np` ≤ 1, the observed spectrum is identical to the radiative background. For cases where the sensor is placed inside the model atmosphere and `np` = 1, the stored position is identical to the sensor position and that position can be used to determinate the radiative background (see below).

refraction [Index] A flag (0 or 1) to indicate if refraction has been considered when determining the path. A value of 1 means that refraction has been considered.

method [String] A string describing the calculation approach. For example, '1D basic geometrical'.

constant [Numeric] The propagation path constant. Such a constant can be assigned to all geometrical paths and for 1D cases (with or without refraction). See Sections 7.6 and [**]. This field can be initiated to a negative value to indicate that the constant is undefined or not yet set. For cases where the constant applies, `ppath_step_agenda` sets this constant at the first call of the agenda if the given value is negative.

pos [Matrix] The position of the propagation path points. This matrix has `np` rows and up to 3 columns. Each row holds a position where column 1 is the radius, column 2 the latitude and column 3 the longitude (cf. Section 3.4.1). The number of columns for 1D and 2D is 2, while for 3D it is 3. The latitudes are stored for 1D cases as these can be of interest for some applications and are useful if the propagation path shall be plotted. The latitudes for 1D give the angular distance to the sensor (see further Section 3.1.1).

The propagation path is stored in reversed order, that is, the position with index 0 is the path point closest to the sensor (and equals the sensor position if it is inside the atmosphere). The full path is stored also for 1D cases with symmetry around a tangent point (in contrast to ARTS-1).

z [Vector] The geometrical altitude for each path position. The length of this vector is accordingly `np`. This is a help variable for plotting and similar purposes. It shall not be used to interpolate the atmospheric fields, as pressure is the main altitude coordinate.

l_step [Vector] The length along the propagation path between the positions in `pos`. The first value is the length between the first and second point etc. For `np` ≥ 2, the length of the vector is `np` - 1. Otherwise it is 0.

gp_p [ArrayOfGridPos] Index position with respect to the pressure grid. The structure for grid positions is described in Section 18.3.

gp_lat [ArrayOfGridPos] As `gp_p` but with respect to the latitude grid.

gp_lon [ArrayOfGridPos] As `gp_p` but with respect to the longitude grid.

los [Matrix] The line-of-sight of the propagation path at each point. The number of rows of the matrix is `np`. For 1D and 2D, the matrix has a single column holding the zenith angle. For 3D there is an additional column giving the azimuth angle. The zenith and azimuth angles are defined in Section 3.4.2. If the radiative background is the cloud box, the last position (in `pos`) and line-of-sight give the relevant information needed when extracting the radiative background from the cloud box intensity field.

background [String] The radiative background for the propagation path. The possible options for this field are 'space', 'blackbody surface', 'cloud box interior' and 'cloud box surface', where the source of radiation should be clear the content of the strings.

tan_pos [Vector] The position of the tangent point. This vector is only set if there exists a tangent point (above the surface level), the length of the vector is otherwise 0. The tangent point is defined as the point with the lowest radius along the path. This means that (the absolute value of) the zenith angle at the tangent point is always 90°. For 2D and 3D this point can deviate from the point with lowest geometrical altitude.

geom_tan_pos [Vector] The position of the geometrical tangent point. This vector is set for all downward observations. Refraction and surface reflections are neglected when calculating this tangent point position. This field is not handled by `ppath_step_agenda`. Definition of the tangent point as for `tan_pos`.

The fields above are set as when the propagation path is determined (inside(`ppath_calc`). Remaining fields, listed below, are set at a later stage:

p [Vector] The pressure for each path position. Length is accordingly `np`. Set by `iy_calc`.

t [Vector] The temperature for each path position. Length is accordingly `np`. Set by `iy_calc`.

vmr [Matrix] The VMR of each gas species at each path position. Set by `iy_calc`.

next_parts [ArrayOfIndex] The index in `ppath_array` of following propagation path parts. Not defined for individual propagation paths. A negative value means that there is no following part. Set by functions where new propagation path are calculated, such as `surfaceCalc`.

Algorithm 2 Outline of the function `ppath_calc`.

```
check consistency of function input
call ppath_start_stepping to set ppath_step
create an array of Ppath structures, ppath_array
add ppath_step to ppath_array
while radiative background not reached do
  call ppath_step_agenda
  if path is at the highest pressure surface then
    radiative background is space
  else if path is at either end point of latitude or longitude grid then
    this is not allowed, issue an runtime error
  end if
  if cloud box is active then
    if path is at the surface of the cloud box then
      radiative background is the cloud box surface
    end if
  end if
  add ppath_step to ppath_array
end while
initialize the WSV ppath to hold found number of path points
copy data from ppath_array to ppath
```

7.4 Structure of implementation

The workspace method for calculating propagation paths is `ppathCalc`, but this is just a getaway function for `ppath_calc`. The main use of `ppathCalc` is to debug and test the path calculations, and that WSM should normally not be part of the control file. Propagation paths, or steps, are generated from inside other functions.

7.4.1 Main functions for clear sky paths

The master function to calculate full clear sky propagation paths is `ppath_calc`. This function is outlined in Algorithm 2. The function can be divided into three main parts, initialization (handled by `ppath_start_stepping`), a repeated call of `ppath_step_agenda` and putting data into the return structure (`ppath`).

The main task of the function `ppath_start_stepping` is to set up `ppath_step` for the first call of `ppath_step_agenda`, which means that the practical starting point for the path calculations must be determined. If the sensor is placed inside the model atmosphere, the sensor position gives directly the starting point. For cases when the sensor is found outside the atmosphere, the point where the path exits the atmosphere must be determined. The exit point can be determined by pure geometrical calculations (see Sections 7.6 and 7.7) as the refractive index is assumed to have the constant value of 1 outside the atmosphere. The problem is accordingly to find the geometrical crossing between the limit of the atmosphere and the sensor line-of-sight (LOS). The function performs further some other tasks, which include:

- For all LOS with a zenith angle $\geq 90°$ the position of the geometrical tangent point is calculated.
- If the sensor is placed inside the model atmosphere
  - Checks that the sensor is placed above the surface level. If not, an error is issued.
  - Checks for 2D and 3D and when the sensor position as at an end point of the latitude or longitude grid, that the LOS is inwards with respect to the atmospheric limit.
  - If the sensor and surface altitudes are equal, and the sensor LOS is downward, the radiative background is set to be the surface. For 2D and 3D, the tilt of the surface radius is considered when determining if the LOS is downward.
  - If the cloud box is active and the sensor position is inside the cloud box, the radiative backsurface is set to be "cloud box interior". All sensor positions on the cloud box surface are for 2D and 3D treated as points inside the box (for simplicity reasons), while for 1D the behavior is as expected.
- If the sensor is placed outside the model atmosphere
  - Checks that the zenith angle is $\geq 90°$. Upward observations are here not allowed.
  - If it is found for 2D and 3D that the exit point of the path not is at the top of the atmosphere, but is either at a latitude or longitude end face of the atmosphere, an error is issued. This problem can not appear for 1D.

For further details, see the code.

7.4.2 Main functions for propagation path steps

Example on workspace methods to calculate propagation path steps are `ppath_stepGeometric` and `ppath_stepRefractionEuler`. All such methods adapt automatically to the atmospheric dimensionality, but the different dimensionalities are handled by separate internal functions. For example, the sub-functions to `ppath_stepGeometric` are `ppath_step_geom_1d`, `ppath_step_geom_2d` and `ppath_step_geom_3d`. See `m_ppath.cc` to get the names of the sub-functions for other propagation path step workspace methods. The variables to describe the atmosphere are compacted for 1D and 2D when handed over to atmospheric dimensionality specific sub-functions. For example, the variable in `ppath_step_geom_2d` for the geoid radius is a vector, while the workspace variable is a matrix.

Many tasks are independent of the algorithm for refraction that is used, or if refraction is considered at all. These tasks are solved by two functions for each atmospheric dimensionality. For 1D the functions are `ppath_start_1d` and `ppath_end_1d`, and the corresponding functions for 2D and 3D are named in the same way. The functions to calculate geometrical path steps are denoted as `do_gridrange_1d`, `do_gridcell_2d` and `do_gridcell_3d`. Paths steps passing a tangent point are handled by a recursive call of the step function. Algorithm 3 summerizes this for geometrical 2D steps.

Algorithm 3 Outline of the function `ppath_step_geom_2d`.

```
call ppath_start_2d
if ppath_step.ppc < 1 then
   calculate the path constant (this is then first path step)
end if
call do_gridcell_2d
call ppath_end_2d
if calculated step ends with tangent point then
   call ppath_step_geom_2d with temporary Ppath structure
   append temporary Ppath structure to ppath_step
end if
```

7.5 General comments

The calculation of propagation paths involves a number of mathematical expressions and they are presented in Sections 7.6 - 7.8. In addition, the path calculations present a number of practical problems. These practical problems are discussed briefly in this section. For further details, see the code.

7.5.1 Numerical precision

The aim here is not to make a complete discussion around the limited numerical accuracy, but just to point out some of the problems caused. We can start by noticing that the precision with which atmospheric positions can be given is about 0.5 m when the numeric type is float and $2{\cdot}10^{-8}$ m for double (assuming that the mantissa has 24 and 48 bits, respectively). The numbers given correspond to the change of the position for a change of 1 bit, in either radius, latitude and longitude. Already these numbers cause problems for the approach taken to calculate propagation paths. For any path along the border of a grid cell, any rounding error in the wrong direction will move the position outside the grid cell, which would lead to a crash of the code without countermeasures.

The values above give the representation precision. The precision will be even poorer if a position is obtained by calculations as numerical problems tend to accumulate. The calculation precision depends on what mathematical expressions that are involved. For example, a radius or length obtained by the Pythagorean relation will have a relatively high uncertainty as the calculations involve taking the square of a radius in the order of 6400 km. It was found that for calculations performed using only float as numeric type, could lead to displacements from the true position up to 10 m. It was first tried to hard-code double as the numerical type for the most critical passages of the calculations, but a total success was not achieved and some code had to be duplicated (to be used with either the float or double option by if-statements for the pre-compiler) to avoid compiler warnings. A step further was then taken, and double is now hard-coded for all internal variables of `ppath.cc`. This deviation from the rule to have an uniform numeric type inside ARTS was introduced to avoid more complicated coding and it has a very small impact on the overall calculation speed. However, this measure will not lead to that the precision of the path calculations will be the same for float and double, as the results will be converted to float between each propagation path step when copied to `ppath_step`.

As pointed out above, the most critical cases are when the path goes along the boundary of a grid cell. This situation is not common for arbitrary observation positions, but it is a standard case for 3D scattering calculations as the starting point for the calculations there is always a crossing point of the atmospheric grids. The solution to this problem is to introduce special treatment for such geometrical paths. For strictly vertical 2D and 3D paths, the latitude, and also longitude for 3D, of the start and end points shall be identical. Paths in 3D with an azimuth angle of 0°or 180° have a constant longitude; the paths are in the north-south plane, and this should also then be valid for the longitude value of the start and end positions of the path step.

The variables connected to different problems associated with the numerical inaccuracy and singularity of mathematical expressions are defined at the top of the file `ppath.cc`. The variables include the accepted tolerance when making asserts in internal functions that the given point is inside the specified grid cell. Another example is the latitude limit to use the special mathematical expressions needed for positions on the poles.

7.5.2 Propagation paths and grid positions

The grid positions are calculated on the same time as the path is determined. The main reason to this is that the grid positions make it possible to quickly determine inside which grid box the path step is found. Without the grid positions, each call of the functions would need a costly search to locate the starting position with respect to the grids. If you are not familiar with grid positions, it is recommended to read Section 18 before you continue here.

The limited numerical accuracy requires some care when setting the grid positions. First of all, rounding errors can give a fractional distance < 0 or > 1 and this must be avoided. The function `gridpos_check_fd` was created for this purpose, and should be called for each grid position. This function just sets all values below 0 to 0 and all value above 1 to 1. In addition, the grid position for the end point of a path step (beside when there is an intersection with the ground) must have one fractional distance of exactly 0 or 1, but this is not ensured by `gridpos_check_fd` and for end points the function `gridpos_force_end_fd` shall also be called.

Some care is needed to determine in which grid range a path step is found. First of all, there exists an ambiguity for the fractional distance at the grid points. It can either be 0 or 1. In addition, if a position is exactly on top of a grid point, the observation direction determines the interesting grid range. As an help to resolve these question there is the function `gridpos2gridrange`. This function takes an argument describing the direction of the line-of-sight with respect to the grids. This argument shall be set to 1 if the viewing direction is towards higher indexes. The direction argument can be set with the following logical expressions, for the different combinations of atmospheric dimensionality and grid of interest:

1D-3D, pressure: $|\psi| \leq 90°$
2D, latitude: $\psi \geq 0°$
3D, latitude: $\omega \leq 90°$
3D, longitude: $\omega \geq 0°$

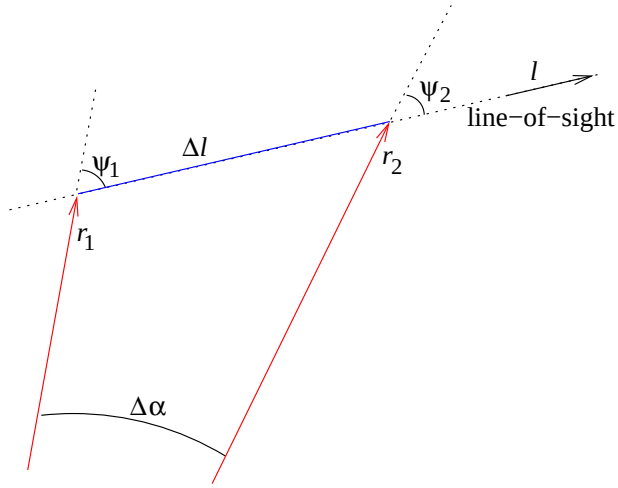

Figure 7.3: The radius (r) and zenith angle (ψ) for two points along the propagation path, and the distance along the path (Δl) and the latitude difference ($\Delta\alpha$) between these points.

7.6 Some basic geometrical relationships for 1D and 2D

This section gives some expressions to determine positions along a propagation path when refraction is neglected. The expressions deal only with propagation path inside a plane, where the latitude angle is the angular distance from an arbitrary point. This means that the expressions given here can be directly applied for 1D and 2D. Some of the expression are also of interest for 3D. The ARTS method for making the calculation of concern is given inside parenthesis above each equation, if not stated explicitly. A part of a geometrical propagation path is shown in Figure 7.3.

The law of sines gives that the product must $r\sin(\psi)$ be constant along the propagation path:

$$p_c = r\sin(\psi) \tag{7.1}$$

where the absolute value is taken for 2D zenith angles as they can for such cases be negative. The propagation path constant, p_c, is determined by the position and line-of-sight of the sensor, a calculation done by the function `geometrical_ppc`. The constant equals also the radius of the tangent point of the path (that is found along an imaginary prolongation of the path behind the sensor if the viewing direction is upwards). The expressions below are based on p_c as the usage of a global constant for the path should decrease the sensitivity to numerical inaccuracies. If the calculations are based solely on the values for the neighboring point, a numerical inaccuracy can accumulate when going from one point to next. The propagation path constant is stored in the field `constant` of `ppath` and `ppath_step`.

The relationship between the distance along the path for an infinitesimal change in radius is here denoted as the geometrical factor, g. If refraction is neglected, valid expressions for the geometrical factor are

$$g = \frac{\mathrm{d}l}{\mathrm{d}r} = \frac{1}{\cos(\psi)} = \frac{1}{\sqrt{1-\sin^2(\psi)}} = \frac{r}{\sqrt{r^2-p_c^2}} \tag{7.2}$$

For the radiative transfer calculations, only the distance between the points, Δl, is of interest, but for the internal propagation path calculations the length from the tangent point (real

or imaginary), l, is used. By integrating Equation 7.2, we get that (`geomppath_l_at_r`)

$$l(r) = \sqrt{r^2 - p_c^2} \tag{7.3}$$

As refraction is here neglected, the tangent point, the point of concern and the center of the coordinate system make up a right triangle and Equation 7.3 corresponds to the Pythagorean relation where p_c is the radius of the tangent point. The distance between two points (Δl) is obtained by taking the difference of Equation 7.3 for the two radii.

The radius for a given l is simply (`geomppath_r_at_l`)

$$r(l) = \sqrt{l^2 + p_c^2} \tag{7.4}$$

The radius for a given zenith angle is simply obtained by rearranging Equation 7.1 (`geomppath_r_at_za`)

$$r(\psi) = \frac{p_c}{sin(\psi)} \tag{7.5}$$

The zenith angle for a given radius is (`geomppath_za_at_r`)

$$\psi(r) = \begin{cases} 180 - \sin^{-1}(p_c/r) & \text{for} \quad 90^\circ < \psi_a \leq 180^\circ \\ \sin^{-1}(p_c/r) & \text{for} \quad 0^\circ \leq \psi_a \leq 90^\circ \\ -\sin^{-1}(p_c/r) & \text{for} \quad -90^\circ \leq \psi_a < 0^\circ \\ \sin^{-1}(p_c/r) - 180 & \text{for} \quad -180^\circ \leq \psi_a < -90^\circ \end{cases} \tag{7.6}$$

where ψ_a is any zenith angle valid for the path on the same side of the tangent point. For example, for a 1D case, the part of the path between the tangent point and the sensor has zenith angles $90^\circ < \psi_a \leq 180^\circ$.

The latitude for a point (`geomppath_lat_at_za`) is most easily determined by its zenith angle

$$\alpha(\psi) = \alpha_0 + \psi_0 - \psi \tag{7.7}$$

where ψ_0 and α_0 are the zenith angle and latitude of some other point of the path. Equation 7.7 is based on the fact that the quantities ψ_1, ψ_2 and $\Delta\alpha$ fulfill the relationship

$$\Delta\alpha = \psi_1 - \psi_2, \tag{7.8}$$

this independently of the sign of the zenith angles. The definitions used here result in that the absolute value of the zenith angle always decreases towards zero when following the path in the line-of-sight direction, that is, when going away from the sensor. It should then be remembered that the latitudes for 1D measures the angular distance to the sensor, and for 2D a positive zenith angle means observation towards higher latitudes.

The radius for a given latitude (`geomppath_r_at_lat`) is obtained by combining Equations 7.7 and 7.5.

7.7 Calculation of geometrical propagations paths

This section describes the calculation of geometrical propagation paths for different atmospheric dimensionalities. That is, the effect of refraction is neglected. These calculations are performed by the workspace method `ppath_stepGeometric`. This method, as all methods for propagation path steps, adjust automatically to the atmospheric dimensionality, but the actual calculations are performed a sub-function for each dimensionality.

7.7.1 1D

The core function for this case is `do_gridrange_1d`. The lowest and highest radius value along the path step is first determined. If the line-of-sight is upwards ($|\psi| \leq 90°$), then the start point of the step gives the lowest radius, and the radius of the pressure surface above gives the highest value. In the case of a downwards line-of-sight, the lowest radius is either the tangent point, the pressure surface below or the surface. The needed quantities to describe the propagation path between the two found radii are calculated by the function `geompath_from_r1_to_r2`, that has the option to introduce more points to fulfill a length criterion between the path points. The mathematics of `geompath_from_r1_to_r2` are given by Equations 7.1 - 7.7.

7.7.2 2D

The definitions given in Sections 3.1.1 results in that for a 2D case the radius of a pressure surface varies linearly from one point of the latitude grid to next. This is the main additional problem to solve, compared to the 1D case. Figure 7.4 gives a schematic description of the problem at hand, which is handled by the internal function `psurface_crossing_2d`.

The law of sine gives the following relationship for the crossing point:

$$\frac{\sin\Theta_p}{r_0 + c\alpha} = \frac{\sin(\pi - \alpha - \Theta_p)}{r_p} \tag{7.9}$$

which can be re-written to

$$r_p \sin(\Theta_p) = (r_0 + c\alpha)(\sin\Theta_p \cos\alpha + \cos\Theta_p \sin\alpha) \tag{7.10}$$

This equation has no analytical solution. A first step to find an approximative solution is to note that α will be limited to relatively small values. For example, if it shall be possible for the angular distance α to reach the value of 3°, the vertical spacing between the pressure surfaces must be about 8 km, while it normally is below 2 km. For angles $\alpha \leq 3°$, the sine and cosine terms can be replaced with the two first terms of their Taylor expansions with a relative accuracy of $< 4 \cdot 10^{-7}$. That is,

$$\begin{aligned} \cos\alpha &\approx 1 - \alpha^2/2 \\ \sin\alpha &\approx \alpha - \alpha^3/6 \end{aligned}$$

Equation 7.10 becomes with these replacements a polynomial equation of order 4:

$$\begin{aligned} 0 &= p_0 + p_1\alpha + p_2\alpha^2 + p_3\alpha^3 + p_4\alpha^4 \\ p_0 &= (r_0 - r_p)\sin\Theta_p \end{aligned} \tag{7.11}$$

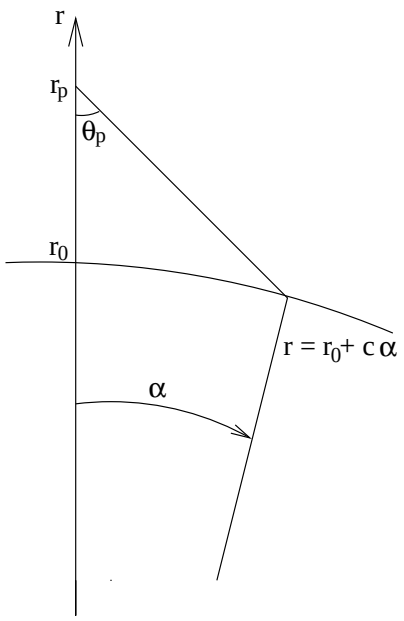

Figure 7.4: Quantities used to describe how to find the crossing between a geometrical propagation path and a tilted pressure surface. The angle α is the angular distance from a reference point on the path. The problem at hand is to find α for the crossing point. The radius of the pressure surface at $\alpha = 0$ is denoted as r_0. The tilt of the pressure surface is c.

$$
\begin{aligned}
p_1 &= r_0 \cos\Theta_p + c \sin\Theta_p \\
p_2 &= -(r_0 \sin\Theta_p)/2 + c\cos\Theta_p \\
p_3 &= -(r_0 \cos\Theta_p)/6 - (c\sin\Theta_p)/2 \\
p_4 &= -(c\cos\Theta_p)/6
\end{aligned}
$$

This equation is solved numerically with the root finding algorithm implemented in the function `poly_root_solve`. Solutions of interest shall not be imaginary.

Geometrical 2D propagation path steps are determined by `do_gridcell_2d`. This function uses `psurface_crossing_2d` to calculate the latitude distance to a crossing of the pressure surface below and above the present path point. If the closest crossing point with the pressure surfaces is outside the latitude range of the grid cell, it is the crossing of the path with the end latitude (in the viewing direction) that is of interest (Figure 7.5).

7.7.3 3D

Geometrical 3D propagation path steps are determined by the function `do_gridcell_3d`. It was first tested to use different analytical expressions to calculate the length between a point and the crossing of some radius, latitude or longitude. However, the expressions found include the trigonometric functions and the squaring of radii, which resulted in a high sensitivity to the numerical inaccuracy. It was found that the numerical problems made the created algorithm impossible to use in practice. Equation 7.20 below is a reminiscence of that work. In addition, no simple solution to the problem of finding the crossing with a tilted 3D pressure surface using the analytical expressions was found.

A straightforward trail-and-error algorithm was then tested (Algorithm 4 and Figure 7.6). The main advantage of the algorithm is that a correction for the shift in position

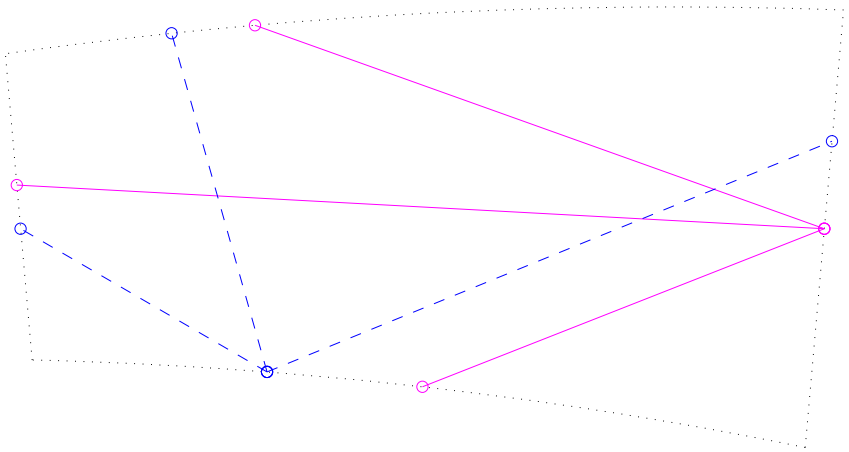

Figure 7.5: Example on propagation path steps starting from a latitude end face (solid lines), or the lower pressure surface (dashed lines), to all other grid cell faces. The distortion of the grid cell from cylinder segment is highly exaggerated compared to a real case. The relationship between vertical and horisontal size deviates also from normal real cases. Typical values for the vertical extension is around 500 m, while the horisontal length is normally > 10 km.

caused by the transformations back and fourth to a cartesian coordinate system can be applied. The correction term assures that the position is not changed for a step of zero length, and is not moved outside the grid cell due to the numerical problems. The algorithm was further found to be sufficiently fast to be accepted. A simple bisection search to find the length of the propagation path step is used. Both the position and the line-of-sight for the other end point of the path step are calculated using a transformation to cartesian coordinates. The cartesian coordinate system used here is defined as:

x-axis is along latitude 0°and longitude 0°

y-axis is along latitude +90°

z-axis is along latitude 0°and longitude +90°

This definition results in the following relationships between the spherical (r, α, β) and cartesian (x, y, z) coordinates

$$\begin{aligned} x &= r\cos(\alpha)\cos(\beta) \\ y &= r\sin(\alpha) \\ z &= r\cos(\alpha)\sin(\beta) \end{aligned} \tag{7.12}$$

and

$$\begin{aligned} r &= \sqrt{x^2 + y^2 + z^2} \\ \alpha &= \arcsin(y/r) \\ \beta &= \arctan(z/x) \qquad \text{(implemented by the atan2 function)} \end{aligned} \tag{7.13}$$

Algorithm 4 The method applied in `do_gridcell_3d` to find the total length of the path step to be calculated. The symbol S signifies here conversion from cartesian to spherical coordinates (Equation 7.13).

calculate the spherical position (x_0, y_0, z_0) and LOS vector $(\mathrm{d}x, \mathrm{d}y, \mathrm{d}z)$
calculate $(r_c, \alpha_c, \beta_c) = S(x_0, y_0, z_0) - (r_0, \alpha_0, \beta_0)$, the position correction term
set $l_{in} = 0$
set $l_{out} = 1$
if LOS is downwards **then**
  calculate length to the tangent point, l_{tan}
else
  set $l_{tan} = 99 \cdot 10^6$ m
end if
while $S(x_0 + l_{out}\mathrm{d}x, y_0 + l_{out}\mathrm{d}y, z_0 + l_{out}\mathrm{d}z) - (r_c, \alpha_c, \beta_c)$ is inside grid cell **do**
  if $l_{out} < l_{tan}$ and $10 l_{out} > l_{tan}$ **then**
    $l_{out} = l_{tan}$ (to assure that tangent point is included in search)
  else
    $l_{out} \leftarrow 10 * l_{out}$
  end if
end while
set $l_{end} = (l_{in} + l_{out})/2$
set accuracy flag to false
while accuracy flag is false **do**
  calculate $(r, \alpha, \beta) = S(x_0 + l_{end}\mathrm{d}x, y_0 + l_{end}\mathrm{d}y, z_0 + l_{end}\mathrm{d}z) - (r_c, \alpha_c, \beta_c)$
  if (r, α, β) is inside grid cell **then**
    $l_{in} = l_{end}$
  else
    $l_{out} = l_{end}$
  end if
  if $(l_{out} - l_{in})$ smaller than specified accuracy **then**
    set accuracy flag to true
  else
    $l_{end} = (l_{in} + l_{out})/2$
  end if
end while
$(r, \alpha, \beta) \leftarrow (r, \alpha, \beta) + (r_c, \alpha_c, \beta_c)$

The functions performing these transformations are `sph2cart` and `cart2sph`.

The first step to transform a line-of-sight, given by the zenith (ψ) and the azimuth (ω) angle, to cartesian coordinates is to determine the corresponding vector with unit length in the spherical coordinate system:

$$\begin{bmatrix} \mathrm{d}r \\ \mathrm{d}\alpha \\ \mathrm{d}\beta \end{bmatrix} = \begin{bmatrix} \cos(\psi) \\ \sin(\psi)\cos(\omega)/r \\ \sin(\psi)\sin(\omega)/(r\cos(\alpha)) \end{bmatrix} \tag{7.14}$$

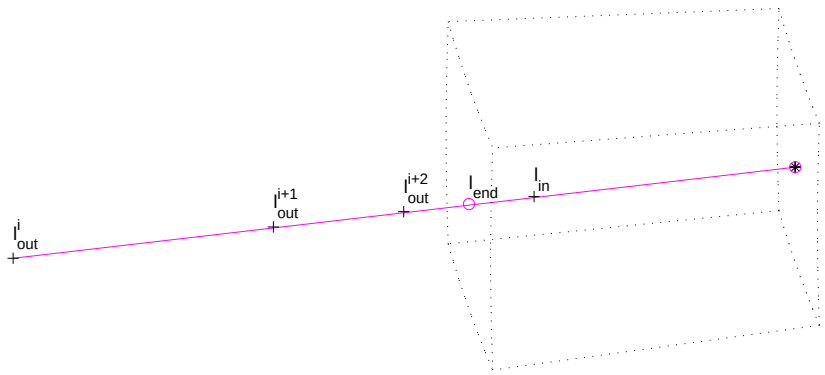

Figure 7.6: Schematic of Algorithm 4. The figure shows two iterations of the algorith to search for the total length of the path step. The asterisk $(*)$ gives the start point for the calculations and the circles $(\circ)$ are the final end points of the path step. The plus signs $(+)$ shows the position of the different lengths tested during the iterations.

This vector is then translated to the cartesian coordinate system as

$$\begin{bmatrix} \mathrm{d}x \\ \mathrm{d}y \\ \mathrm{d}z \end{bmatrix} = \begin{bmatrix} \cos(\alpha)\cos(\beta) & -r\sin(\alpha)\cos(\beta) & -r\cos(\alpha)\sin(\beta) \\ \sin(\alpha) & r\cos(\alpha) & 0 \\ \cos(\alpha)\sin(\beta) & -r\sin(\alpha)\sin(\beta) & r\cos(\alpha)\cos(\beta) \end{bmatrix} \begin{bmatrix} \mathrm{d}r \\ \mathrm{d}\alpha \\ \mathrm{d}\beta \end{bmatrix} \quad (7.15)$$

Note that the radial terms (r) in Equations 7.14 and 7.15 cancel each other. These calculations are performed in `poslos2cart`. Special expressions must be used for positions at the north and south pole (see the code) as the azimuth angle has there a special definition (Section 3.4.2).

The cartesian position of a point along the geometrical path at a distance l is then simply

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 + l\mathrm{d}x \\ y_1 + l\mathrm{d}y \\ z_1 + l\mathrm{d}z \end{bmatrix} \quad (7.16)$$

The cartesian viewing vector $[\mathrm{d}x, \mathrm{d}y, \mathrm{d}z]^T$ is constant along a geometrical path. The new position is converted to spherical coordinates by Equation 7.13 and the new spherical viewing vector is calculated as

$$\begin{bmatrix} \mathrm{d}r \\ \mathrm{d}\alpha \\ \mathrm{d}\beta \end{bmatrix} = \begin{bmatrix} \cos(\alpha)\cos(\beta) & \sin(\alpha) & \cos(\alpha)\sin(\beta) \\ -\sin(\alpha)\cos(\beta)/r & \cos(\alpha)/r & -\sin(\alpha)\sin(\beta)/r \\ -\sin(\beta)/(r\cos(\alpha)) & 0 & \cos(\beta)/(r\cos(\alpha)) \end{bmatrix} \begin{bmatrix} \mathrm{d}x \\ \mathrm{d}y \\ \mathrm{d}z \end{bmatrix} \quad (7.17)$$

which is converted to a zenith and azimuth angle as

$$\begin{aligned} \psi &= \arccos(\mathrm{d}r) \\ \omega &= \arccos(r\mathrm{d}\alpha/\sin(\psi)), \qquad \text{for} \quad \mathrm{d}\beta >= 0 \\ \omega &= -\arccos(r\mathrm{d}\alpha/\sin(\psi)), \qquad \text{for} \quad \mathrm{d}\beta < 0 \end{aligned} \quad (7.18)$$

Special expressions must be used for positions at the north and south pole (see the code) as the azimuth angle has there a special definition (Section 3.4.2). These calculations are performed in `cart2poslos`.

For sensor positions outside the atmosphere, the calculations made in `ppath_start_stepping` involve the problem of finding the position where the path leaves the atmosphere. This position is found by an iterative search. The maximum radius of the uppermost pressure surface is taken as first guess for the radius of the exit point. The exit latitude and longitude for this radius is determined (as discussed below), and the radius for the top of the atmosphere for the found position is used as radius for next iteration. This procedure is repeated until the change from one iteration to next for both latitude and longitude is smaller than $1{\cdot}10^{-6}$. The exit position for a given radius, r, is found by solving the following equation system:

$$\begin{aligned} r\cos(\alpha)\cos(\beta) &= x + l\mathrm{d}x \\ r\sin(\alpha) &= y + l\mathrm{d}y \\ r\cos(\alpha)\sin(\beta) &= z + l\mathrm{d}z \end{aligned} \tag{7.19}$$

where (x, y, z) is the position of the sensor, $(\mathrm{d}x, \mathrm{d}y, \mathrm{d}z)$ the sensor LOS, and l, α and β are the variables to be determined. The first step is to determine the distance l to the exit point, which is found by adding the square of all three equations:

$$r^2 = (x + l\mathrm{d}x)^2 + (y + l\mathrm{d}y)^2 + (z + l\mathrm{d}z)^2 \tag{7.20}$$

Once l is determined, the latitude and longitude are easily calculated by Equations 7.16 and 7.13. These calculations are implemented in the function `psurface_crossing_3d`. Similar expressions were derived to find the position for the crossing of a given latitude or longitude but those expressions were removed from the code as they are not used with present algorithms.[1]

7.8 Refraction with simple Euler scheme

Refraction affects the radiative transfer in several ways. The distance through a layer of a fixed vertical thickness will be changed, and for a limb sounding observation the tangent point is moved both vertically and horizontally. If the atmosphere is assumed to be horizontally stratified (1D), a horizontal displacement is of no importance but for 2D and 3D calculations this effect must be considered. For limb sounding and a fixed zenith angle, the tangent point is moved downwards compared to the pure geometrical case (Figure 7.7), resulting in that inclusion of refraction in general gives higher intensities. However, the propagation path is still symmetric around tangent and surface points.

The refraction causes a bending of the path, which gives a deviation from the geometrical approximation of propagation along a straight line. The bending of the path is obtained by the relationship

$$\frac{\mathrm{d}x}{\mathrm{d}l} = \frac{1}{n}\left(\frac{\partial n}{\partial y}\right)_x \tag{7.21}$$

[1] The expressions mentioned can be extracted from the function `gridcell_crossing_3d` in ARTS version 1-1-440.

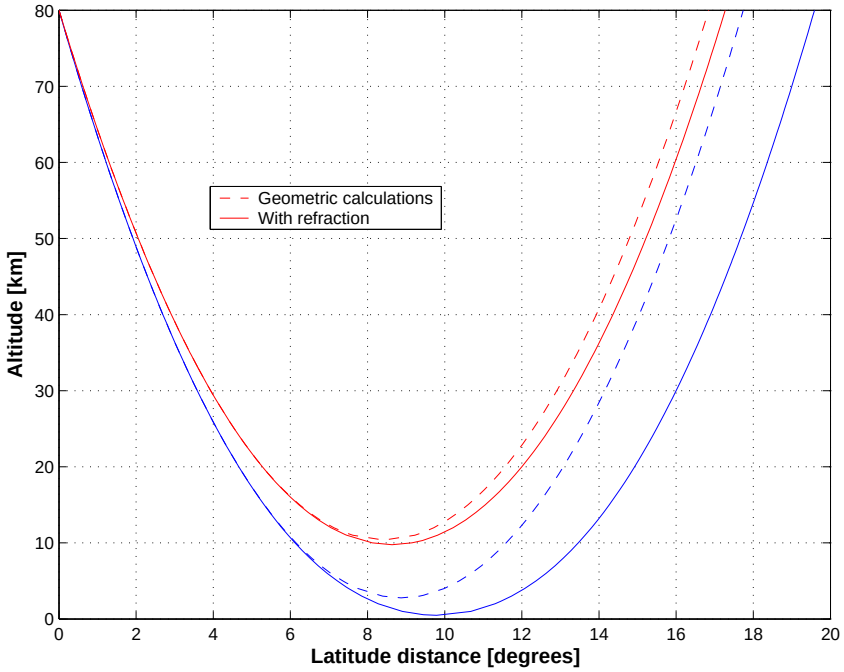

Figure 7.7: Comparison of propagation paths calculated geometrically and with refraction considered, for the same zenith angle of the sensor line-of-sight. The figure include two pair of paths, with refracted tangent altitude of about 0 and 10 km, respectively. The horisontal coordinate is the latitude distance from the point where the path exits the model atmosphere (at 80 km). The model atmosphere used had a spherical symmetry (that is, 1 D case, but the calculations were performed in 2D mode).

where x is the direction of propagation, l the distance along the path, n the refractive index[2], and y is the coordinate perpendicular to the path. See further Section 9.4 in *Rodgers* [2000].

The workspace method `ppath_stepRefractionEuler` takes refraction into consideration by probably the most simple (from the viewpoint of implementation) algorithm possible. This does not mean that it is the best way to consider refraction, it is rather inefficient regarding computational burden, and if the step length for the ray tracing (see below) is made very small, the result can be completely wrong due to numerical problems.

The approach taken in `ppath_stepRefractionEuler` is to take a geometrical ray tracing step from the present point of the path (and in the direction of present line-of-sight). Refraction is considered only when the line-of-sight at the new point is determined (Figure 7.8). The found line-of-sight is used to calculate the next ray tracing step etc. This can be seen as an Euler solution to the differential problem given by Equation 7.21. The main difference between handling 1D, 2D or 3D cases is how the line-of-sight for the new point is corrected to compensate for the bending due to refraction. The calculation of propagation paths including the effect of refraction is often denoted as ray tracing.

The length of the calculation steps is set by the generic input `lraytrace`. This length

[2]The refractive index is here assumed to have no imaginary part

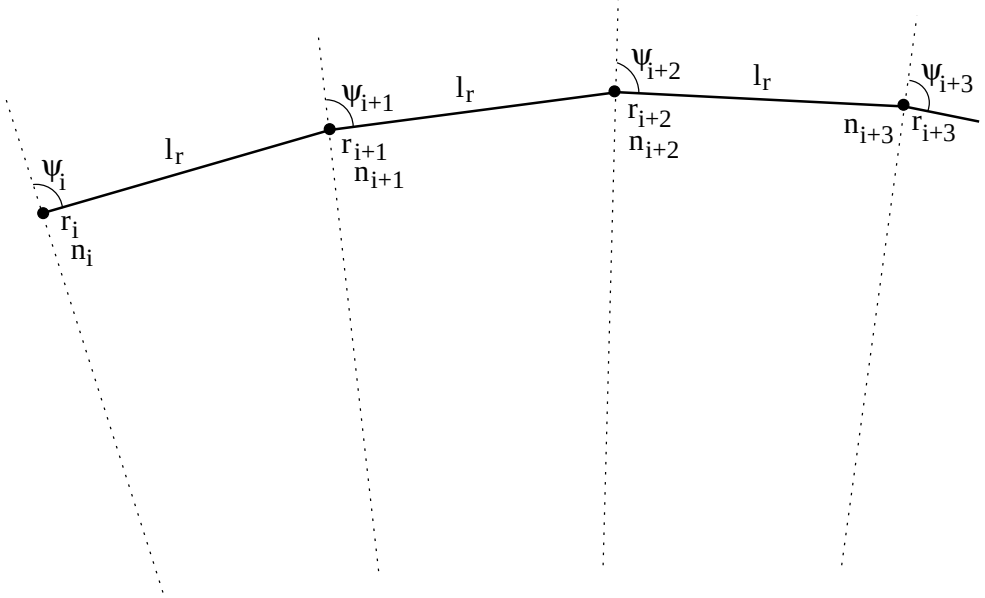

Figure 7.8: Schematic of the Euler ray tracing scheme. The ray tracing step length is l_r.

shall not be confused with the final distance between the points that define the path, which is controlled by the generic input `lmax`. The path is first determined in steps of `lraytrace`. The found ray tracing points are then used for an interpolation to create a path step defined exactly as for geometrical calculations. The normal situation is that the ray tracing step length is considerably shorter than the final spacing between the path points. Suitable values for `lraytrace` have not yet been investigated in detail, but for limb sounding values in around 1 - 10 km should be appropriate. Shorter ray tracing steps (down to a level where rounding errors will start to have an impact) will of course give a propagation path more accurately determined, but on the cost of more time consuming calculations.

7.8.1 1D

When determining the propagation path through the atmosphere geometrical optics can be applied because the change of the refractive index over a wavelength can be neglected. Applying Snell's law to the geometry shown in Figure 7.9 gives

$$n_i \sin(\psi_i) = n_{i+1} \sin(\psi_{i'}) \tag{7.22}$$

Using the same figure, the law of sines gives the relationship

$$\frac{\sin(\psi_{i+1})}{r_i} = \frac{\sin(180^\circ - \psi'_{i+1})}{r_{i+1}} = \frac{\sin(\psi_{i'})}{r_{i+1}} \tag{7.23}$$

By combining the two equations above, the Snell's law for a spherical atmosphere (that is, 1D cases) is derived [e.g. *Kyle*, 1991; *Balluch and Lary*, 1997]:

$$p_c = r_i n_i \sin(\psi_i) = r_{i+1} n_{i+1} \sin(\psi_{i+1}) \tag{7.24}$$

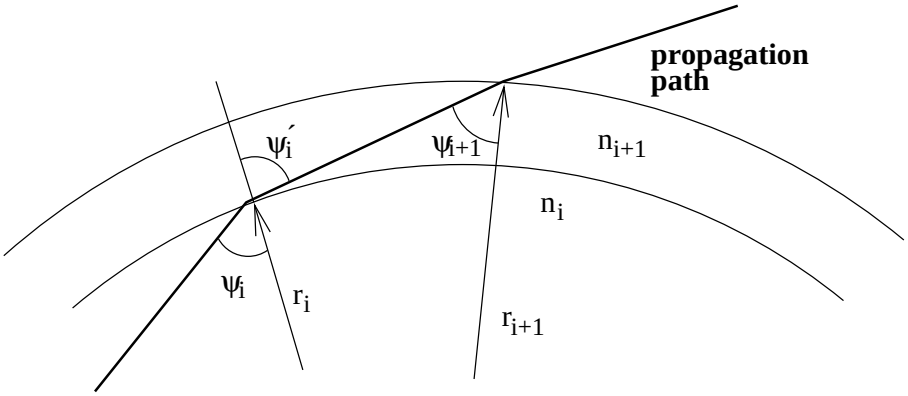

Figure 7.9: Geometry to derive Snell's law for a spherical atmosphere.

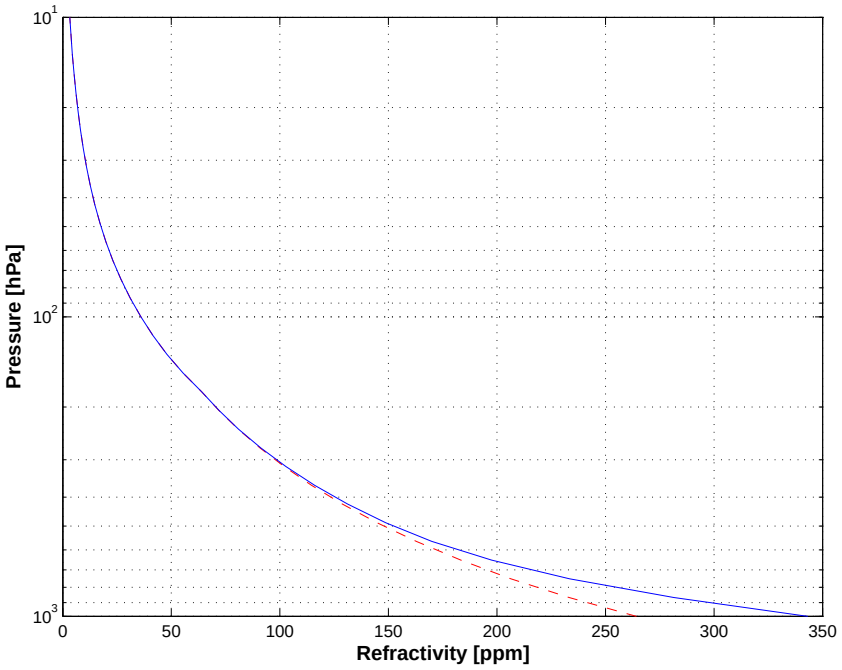

Figure 7.10: Vertical variation of refractivity $(n-1)\cdot 10^6$. Calculated for a mid-latitude summer climatology (FASCODE), where the dashed line is for a completely dry atmosphere, and the solid line includes also contribution from water vapour.

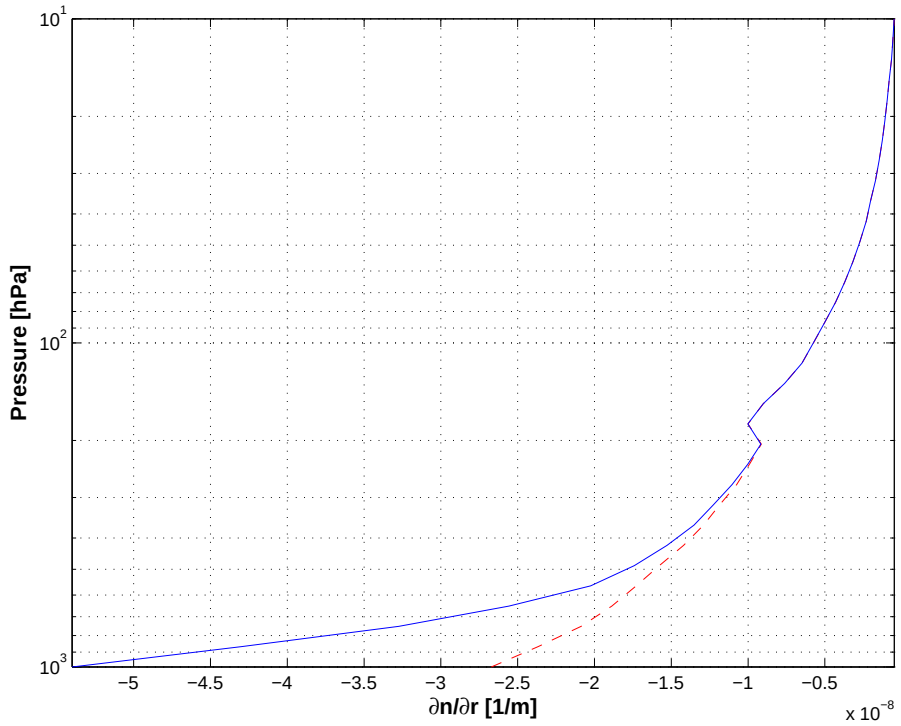

Figure 7.11: Vertical gradient of the refractive index. Calculated for a mid-latitude summer climatology (FASCODE), where the dashed line is for a completely dry atmosphere, and the solid line includes also contribution from water vapour.

where c is a constant. With other words, the Snell's law for spherical atmospheres states that the product of n, r and $\sin(\psi)$ is constant along the propagation path. It is noteworthy that with $n = 1$, Equations 7.1 and 7.24 are identical.

The Snell's law for a spherical atmosphere makes it very easy to determine the zenith angle of the path for a given radius. A rearrangement of Equation 7.24 gives

$$\psi = \arcsin(rn/p_c) \tag{7.25}$$

This relationship makes it possible to handle refraction for 1D without calculating any gradients of the refractive index, which is needed for 2D and 3D. These calculations are implemented in the function `raytrace_1d_linear_euler`. Figure 7.10 shows the vertical variation of the refractive index.

7.8.2 2D

Equation 7.21 expressed in polar coordinates is [*Rodgers*, 2000, Eq. 9.30]

$$\frac{\mathrm{d}(\alpha+\psi)}{\mathrm{d}l} = -\frac{\sin\psi}{n}\left(\frac{\partial n}{\partial r}\right)_\alpha + \frac{\cos\psi}{nr}\left(\frac{\partial n}{\partial \alpha}\right)_r \tag{7.26}$$

If the gradients are zero (corresponding to the geometrical case) we find that the sum of the zenith angle and the latitude is constant along a 2D geometrical path, which is also made

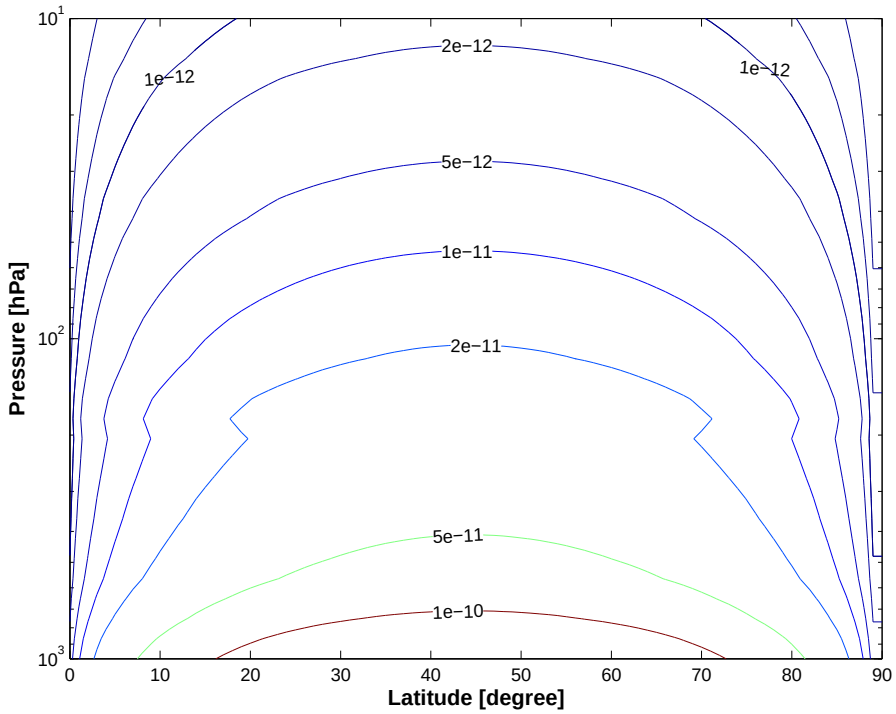

Figure 7.12: Latitude gradient of the refractive index due to varying radius of the geoid. The gradient is given as the change in refractive index over 1 m, which allows direct comparison with the values in Figure 7.11e. The wet atmosphere from Figure 7.11 was used for all latitudes, and the the plotted gradient is only caused by the fact that the radius of the geoid is not constant. The gradient is positive on the southern hemisphere (shown), and negative on the northern hemisphere.

clear by Equation 7.7. The geometrical zenith angle at ray tracing point $i+1$ is accordingly $\psi_{i+1} = \psi_i - (\alpha_{i+1} - \alpha_i)$. If then also the refraction is considered, we get the following expression:

$$\psi_{i+1} = \psi_i - (\alpha_{i+1} - \alpha_i) + \frac{l_g}{n_i}\left[-\sin\psi_i \left(\frac{\partial n}{\partial r}\right)_{\alpha_i} + \frac{\cos\psi_i}{r_i}\left(\frac{\partial n}{\partial \alpha}\right)_{r_i}\right] \tag{7.27}$$

The gradients of the refractive index for 2D are calculated by the function `refr_gradients_2d`. This function returns the gradients as the change of the refractive index over 1 m. The conversion for the latitude gradient corresponds to the $1/r$ term found in Equation 7.27, and this term is accordingly left out in `raytrace_2d_linear_euler`, which is the function of this section.

The radial and latitudinal gradients of the refractive index are calculated in pure numerical way, by shifting the position slightly from the position of concern. Figures 7.11 and 7.12 show example on gradients of the refractive index.

7.8.3 3D

For 3D, the geomtrical expressions are used to calculate the geometrical zenith and azimuth angles at the end of the ray tracing step. Following the methodology for 2D, the geometrical zenith and azimuth angles are then corrected to incorporate the influence of refraction. The zenith angle is calculated as

$$\begin{aligned}\psi_{i+1} &= \psi_g - \frac{l_g \sin\psi_i}{n_i}\left(\frac{\partial n}{\partial r}\right)_{(\alpha_i,\beta_i)} + \\ &+\frac{l_g \cos\psi_i}{r_i n_i}\left[\cos\omega_i\left(\frac{\partial n}{\partial \alpha}\right)_{(r_i,\beta_i)} + \frac{\sin\omega_i}{\cos\alpha_i}\left(\frac{\partial n}{\partial \beta}\right)_{(r_i,\alpha_i)}\right]\end{aligned} \tag{7.28}$$

where ψ_g is the zenith angle obtained from the geometrical expressions. In similar manner, the geometrical azimuth angle, ω_g, is corrected as

$$\omega_{i+1} = \omega_g + \frac{l_g \sin\psi_i}{r_i n_i}\left[-\sin\omega_i\left(\frac{\partial n}{\partial \alpha}\right)_{(r_i,\beta_i)} + \frac{\cos\omega_i}{\cos\alpha_i}\left(\frac{\partial n}{\partial \beta}\right)_{(r_i,\alpha_i)}\right] \tag{7.29}$$

This expression, slightly modified, is found in `raytrace_3d_linear_euler`. The terms of Equation 7.29 missing in that function, are part of `refr_gradients_3d` to convert the gradients to the same unit. The longitude gradient is converted to the unit [1/m] by multiplication with the term $1/(r\cos\alpha)$.

7.9 Geoid ellipsoids and geodetic datums

This section defines the geoid ellipsoid and discusses related issues. The geoid is introduced in Section 3.1.4. The workspace variable representing the geoid is `r_geoid`.

7.9.1 Geoid ellipsoids

All geodetic datums are based on a reference ellipsoid. The ellipsoid is rotationally symmetric around the north-south axis. That is, the ellipsoid radius has no longitude variation, it is only a function of latitude. The ellipsoid is described by an equatorial radius, r_e, and a polar radius, r_p. These radii are indicated in Figure 7.13. The radius of the ellipsoid for a given latitude is

$$r_\odot(\alpha) = \sqrt{\frac{r_e^2 r_p^2}{r_e^2\sin^2\alpha + r_p^2\cos^2\alpha}} \tag{7.30}$$

The radius given by Equation 7.30 can be directly applied for 2D and 3D cases. On the other hand, for 1D cases the reference geoid is by definition a sphere and the radius of this sphere shall be selected in such way that it represents the local shape of a reference ellipsoid. This is achieved by setting $r_\odot$ to the radius of curvature of the ellipsoid. The curvature radius differs from the local radius except at the equator and an east-west direction. For example, at the equator and a north-south direction, the curvature radius is smaller then the local radius, while at the poles (for all directions) it is greater (see further Figure 7.14).

The curvature radius, r_c, of an ellipsoid is [*Rodgers*, 2000]

$$r_c = \frac{1}{r_{ns}^{-1}\cos^2\alpha + r_{ew}^{-1}\sin^2\alpha} \tag{7.31}$$

where r_{ns} and r_{ew} are the north-south and east-west curvature radius, respectively,

$$r_{ns} = r_e^2 r_p^2 (r_e^2\cos^2\omega + r_p^2\sin^2\omega)^{-\frac{3}{2}} \tag{7.32}$$

$$r_{ew} = r_e^2 (r_e^2\cos^2\omega + r_p^2\sin^2\omega)^{-\frac{1}{2}} \tag{7.33}$$

The azimuth angle, ω, is defined in Section 3.4.2. The latitude and azimuth angle to apply in Equations 7.31 - 7.33 shall rather be valid for a middle point of the propagation paths (such as some tangent point), instead of the sensor position.

7.9.2 Geocentric and geodetic latitudes

The fact that the geoid is an ellipsoid, instead of a sphere, opens up for the two different definitions of the latitude. The geocentric latitude, which is the the one used here, is the angle between the equatorial plane and the vector from the coordinate system center to the position of concern. The geodetic latitude is also defined with respect to the equatorial plane, but the angle to the normal to the reference ellipsoid is considered here, as shown in Figure 7.13. It could be mentioned that a geocentric latitude does not depend on the geoid ellipsoid used, while the geodetic latitudes change if another reference ellipsoid is selected. An approximative relationship between the geodetic (α^*) and geocentric (α) latitudes is [*Montenbruck and Gill*, 2000]

$$\alpha^* = \alpha + f\,\sin(2\alpha) \tag{7.34}$$

where f is the flattening of the ellipse:

$$f = \frac{r_e - r_p}{r_e} \tag{7.35}$$

The value of f for the Earth is about 1/298.26. This means that the largest differences between α and α^* are found at mid-latitudes and the maximum value is about 12 arc-minutes.

The zenith and nadir directions shall normally be defined to follow the normal to the reference ellipsoid, but, if nothing else is mentioned, these directions are here treated to go along the vector the center of the coordinate system, as indicated in Figure 7.13. This latter definition is preferred as it results in that a propagation path in the zenith/nadir direction can be described by a single latitude and longitude value. The difference in geometrical altitude when using these two possible definitions on the zenith direction is proportional to the deviation between geocentric and geodetic latitude (Equation 7.34). For an altitude of 100 km around $\alpha = 45°$, the difference is about 350 m.

7.9.3 Geodetic datums

Table 7.1 gives the equatorial and polar radii of the reference ellipsoid for the geodetic datums handled by ARTS.

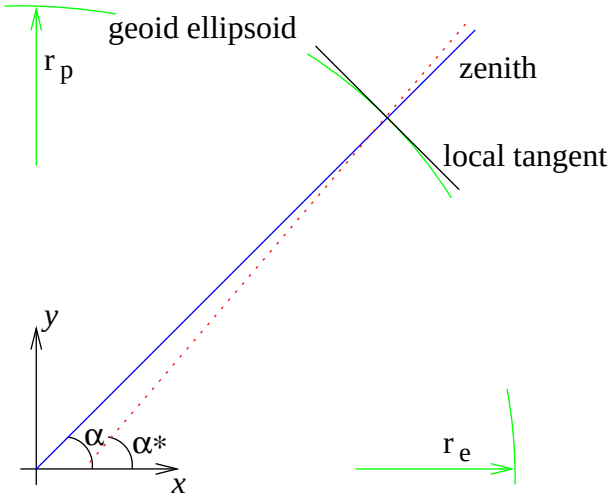

Figure 7.13: Definition of the ellipsoid radii, r_e and r_p, geocentric latitude, α, and geodetic latitude, α^*. The dotted line is the normal to the local tangent of the geoid ellipsoid. The zenith and nadir directions, and geometrical altitudes, are here defined to follow the solid line.

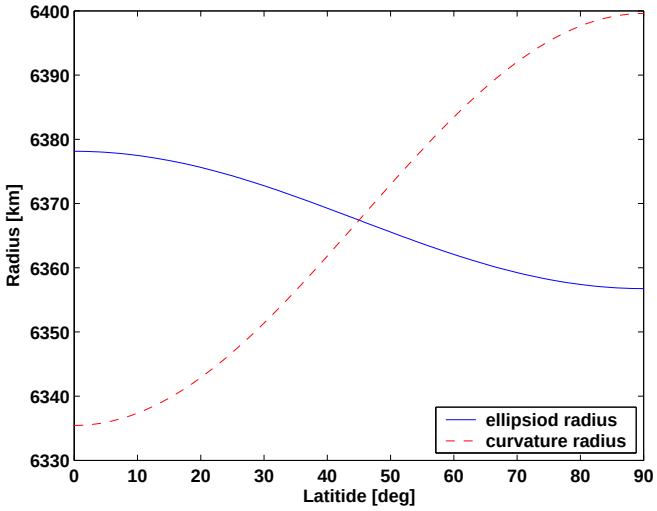

Figure 7.14: The ellipsoid radius ($r_\odot$) and curvature radius (r_c) for the WGS-84 reference ellipsoid. The curvature radii are valid for the north-south direction.

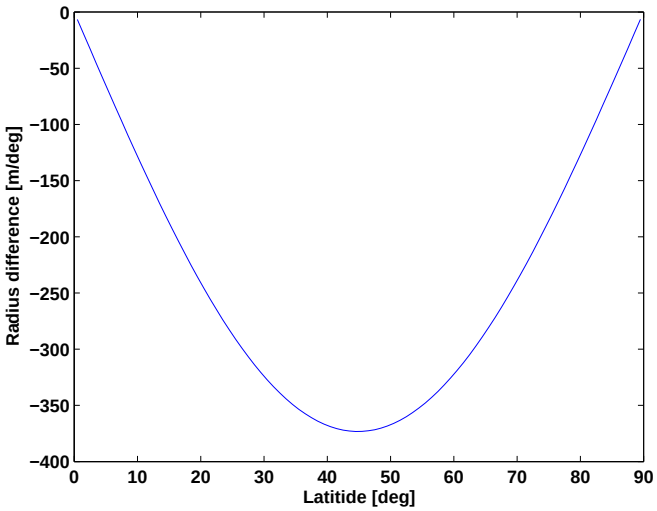

Figure 7.15: The change of the WGS-84 ellipsoid radius for 1° latitude differences.

Datum	r_e	r_p	$1/f$	Reference
WGS-84	6378.137 km	*6356.752 km*	298.2572235	*Montenbruck and Gill* [2000]

Table 7.1: Equatorial and polar radius of reference ellipsoids. Values given as *italic* are derived by the other two values and Equation 7.35.

7.10 Control file examples

Some examples on how the geoid radius and the surface altitude can be set:

```
# Set the geoid to model WGS84 for 2D and 3D.
r_geoidWGS84

# For 1D, *lat_1d* and *meridian_angle_1d* must be specified
# to define the position and direction for which the curvature
# radius shall be extracted.
NumericSet( lat_1d, 45 )
NumericSet( meridian_angle_1d, 0 )
r_geoidWGS84

# Set the geoid to be spherical, with a radius of 6370 km
r_geoidSpherical( r_geoid, atmosphere_dim, lat_grid, lon_grid,
                  6370e3 )

# Set the geoid to be spherical, with standard radius (defined
# in arts.h)
r_geoidSpherical( r_geoid, atmosphere_dim, lat_grid, lon_grid, -1 )

# Set a constant surface altitude of 1 km
ncolsGet( ncols, r_geoid )
nrowsGet( nrows, r_geoid )
MatrixSetConstant( z_surface, nrows, ncols, 1e3 )
```

Different possibilities for the `ppath_step_agenda`:

```
# Select geometric calculations, with no length criterion
# for path points.
AgendaSet( ppath_step_agenda ) {
  ppath_stepGeometric( ppath_step, atmosphere_dim, p_grid, lat_grid,
                       lon_grid, z_field, r_geoid, z_surface, -1 )
}

# Consider refraction. The ray tracing step length is 2 km
# and the length criterion for path points is 10 km.
```

```
AgendaSet( refr_index_agenda ) {
  refr_indexThayer
}
AgendaSet( ppath_step_agenda ) {
  ppath_stepRefractionEuler( ppath_step, rte_pressure,
                             rte_temperature, rte_vmr_list,
                             refr_index, refr_index_agenda,
                             atmosphere_dim, p_grid, lat_grid,
                             lon_grid, z_field, t_field,
                             vmr_field, r_geoid, z_surface,
                             10e3,
                             2e3 )
}
```

Chapter 8

Surface emission and reflections

An introduction to the treatment of surface emission and reflections is given in Section 3.5.5. The methods developed to handle different surface properties all set the variables `surface_emission`, `surface_los` and `surface_rmatrix`.

Let us start with a simple example in order to explain the usage of these workspace variables. We will here assume that all downwelling radiation is reflected. This assumption is made for all polarisation states. We assume further a 1D simulation, that the downwelling radiation shall be calculated for nine zenith angles and that all downwelling directions contribute equally (which is not a realistic assumption). The relevant workspace variables should then be set as follows:

`surface_emission`: A matrix (of correct size) of zeros.

`surface_los`: A vector of length 9, covering the zenith angle range. A possible choice would be [5,15,25,...,85].

`surface_rmatrix`: Each reflection matrix is a diagonal matrix with the value 1/9 throughout on the diagonal. That is, all elements with index (:,:,i,i) is 1/9. Size matching `surface_los`, `f_grid` and `stokes_dim`

8.1 The dielectric constant and the refractive index

The properties of a material are reported either as the relative dielectric constant, ϵ, or the refractive index, n. Both these quantities can be complex and are related as

$$n = \sqrt{\epsilon}. \tag{8.1}$$

8.2 Relating reflectivity and emissivity

Kirchoff's law applied to thermodynamics states that under conditions of local thermodynamic equilibrium, thermal emission has to be equal to absorption [*Ulaby et al.*, 1981, page

History

050613 First version finished by Patrick Eriksson.

215]. This is a consequence of the fact that there must exist a radiation equilibrium between an object and its surrounding, if it is surrounded by a blackbody having the same temperature (with no physical contact).

Thermodynamic equilibrium can be assumed for natural surfaces, as long as there exist no strong temperature gradients. The Kirchoff law can then be used to relate the reflectivity and emissivity of a surface. For rough surfaces the scattering properties must be integrated over the half sphere (above the surface) to determine the emissivity [see e.g. *Ulaby et al.*, 1981, Eq. 4.186]. For specular reflections (defined below) and scalar radiative transfer calculations, the emissivity e is

$$e = 1 - r, \tag{8.2}$$

where r is the reflective (power reflection coefficient) of the surface. Equation 8.2 is valid for each polarisation state individually [*Ulaby et al.*, 1981, Eq. 4.190a].

We have then that

$$I^{\mathrm{up}} = I^{\mathrm{down}} r + (1 - r)B, \tag{8.3}$$

where I^{up} is upwelling radiation, I^{down} is downwelling radiation and B is the magnitude of blackbody radiation. As expected, if $I^{\mathrm{down}} = B$, also I^{up} equals B. Expressing the last observation using vector nomenclature gives

$$\begin{bmatrix} B \\ 0 \\ 0 \\ 0 \end{bmatrix} = \mathbf{R} \begin{bmatrix} B \\ 0 \\ 0 \\ 0 \end{bmatrix} + \mathbf{b}, \tag{8.4}$$

where $\mathbf{R}$ is the matrix (4 x 4) correspondence to the scalar reflectivity, describing the properties of the surface reflection. The vector $\mathbf{b}$ is the surface emission, that can be expressed as

$$\mathbf{b} = (\mathbf{1} - \mathbf{R}) \begin{bmatrix} B \\ 0 \\ 0 \\ 0 \end{bmatrix}, \tag{8.5}$$

where $\mathbf{1}$ is the identity matrix.

8.3 Specular reflections

If the surface is sufficiently smooth, radiation will be reflected/scattered only in the complementary angle, specular reflection. Required smoothness for assuming specular reflection is normally estimated by the Rayleigh criterion:

$$\Delta h < \frac{\lambda}{8 \cos \theta_1} \tag{8.6}$$

where Δh is the root mean square variation of the surface height, λ the wavelength and θ_1 the angle between the surface normal and the incident direction of the radiation. The criterion can also be defined with the factor 8 replaced with a lower integer number.

The complex reflection coefficient for the amplitude of the electromagnetic wave for vertical (R_v) and horizontal (R_v) polarisation is for a flat surface (if the relative magnetic permeability (μ_r) of both media is 1) given by the Fresnel equations:

$$R_v = \frac{n_2 \cos\theta_1 - n_1 \cos\theta_2}{n_2 \cos\theta_1 + n_1 \cos\theta_2} \tag{8.7}$$

$$R_h = \frac{n_1 \cos\theta_1 - n_2 \cos\theta_2}{n_1 \cos\theta_1 + n_2 \cos\theta_2} \tag{8.8}$$

where n_1 is refractive index for the medium where the reflected radiation is propagating, θ_1 is the incident angle (measured from the local surface normal) and n_2 is the refractive index of the reflecting medium. The angle θ_2 is the propagation direction for the transmitted part, and is given by Snell's law:

$$\mathrm{Re}(n_1)\sin\theta_1 = \mathrm{Re}(n_2)\sin\theta_2. \tag{8.9}$$

where $\mathrm{Re}(\cdot)$ denotes the complex real part. For cases where medium 1 is air, n_1 can (in this context) be set to 1.

The power reflection coefficients are converted to an intensity reflection coefficient as

$$r = |R|^2, \tag{8.10}$$

where $|\cdot|$ denotes the absolute value. Note that R can be complex, while r is always real.

The surface reflection can be seen as a scattering event and Section 22.5 can be used to derive the reflection matrix values. The scattering amplitude functions of Equation 22.95 are simply

$$S_2 = R_v, \tag{8.11}$$

$$S_1 = R_h, \tag{8.12}$$

$$S_3 = S_4 = = 0. \tag{8.13}$$

This leads to that the transformation matrix for a specular surface reflection is (compare to *Liou* [2002, Sec. 5.4.3])

$$\mathbf{R} = \begin{bmatrix} \frac{r_v+r_h}{2} & \frac{r_v-r_h}{2} & 0 & 0 \\ \frac{r_v-r_h}{2} & \frac{r_v+r_h}{2} & 0 & 0 \\ 0 & 0 & \frac{R_h R_v^* + R_v R_h^*}{2} & i\frac{R_h R_v^* - R_v R_h^*}{2} \\ 0 & 0 & i\frac{R_v R_h^* - R_h R_v^*}{2} & \frac{R_h R_v^* + R_v R_h^*}{2} \end{bmatrix}. \tag{8.14}$$

If the downwelling radiation is unpolarised, the reflected part of the upwelling radiation is

$$\mathbf{R}\begin{bmatrix} I \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} I(r_v + r_h)/2 \\ I(r_v - r_h)/2 \\ 0 \\ 0 \end{bmatrix}. \tag{8.15}$$

as expected.

If $\mathbf{R}$ is given by Equation 8.14, Equation 8.5 gives that the surface emission is

$$\begin{bmatrix} B\left(1 - \frac{r_v+r_h}{2}\right) \\ B\frac{r_h-r_v}{2} \\ 0 \\ 0 \end{bmatrix}. \tag{8.16}$$

In the case of specular reflections, surface_los shall of course be set to have the length 1. The specular direction is calculated by the internal function surface_specular_los[1]. Equations 8.14 and 8.16 give the values to put into surface_rmatrix and surface_emission. A general method for the specular case (with internal models of the dielectric properties of different media) is surfaceFlat, while surfaceSimple and surfaceBlackbody treat some special cases.

8.4 Control file examples

The agenda surface_prop_agenda sets the surface properties. A simple example where specular reflection is assumed and the surface emissivity is 0.9:

```
AgendaSet( surface_prop_agenda ){
   InterpAtmFieldToRteGps( surface_skin_t, t_field )
   NumericSet( surface_emissivity, 0.9 )
   surfaceSimple
}
```

To read surface properties from files:

```
AgendaSet( surface_prop_agenda ){
   Ignore( rte_gp_p )
   Ignore( rte_gp_lat )
   Ignore( rte_gp_lon )
   Ignore( rte_los )
   ReadXML( surface_los, "surface_los.xml" )
   ReadXML( surface_rmatrix, "surface_rmatrix.xml" )
   ReadXML( surface_emission, "surface_emission.xml" )
}
```

[1] Any tilt of the surface is neglected when determining the specular direction. If there would be any need to consider surface tilt, almost complete code for this task existed in surface_specular_los but was removed in version 1-1-876. The code can be obtained by e.g. checking out version 1-1-875.

Chapter 9

Clear sky radiative transfer

This chapter deals with the "clear sky" part of ARTS. That is, the only physical features to consider are absorption and emission. A simulation can consists solely of clear sky calculations. A clear sky calculations can also treat the radiative transfer from the surface or the cloud box, to the sensor. How these calculations shall be performed is set by defining `rte_agenda`.

So far only unpolarised absorption (absorption does not depend on polarisation state) is treated. Some steps towards handling Zeeman splitting (a feature affecting oxygen absorption) have been taken, but the work has not yet been finished.

9.1 The vector radiative transfer equation

The complete vector radiative transfer, including scattering, is given by Equation 21.35. If scattering can be neglected, the equation can be written as

$$\frac{\mathrm{d}\mathbf{I}}{\mathrm{d}s} = -\mathbf{K}\mathbf{I} + \mathbf{a}B, \tag{9.1}$$

where $\mathbf{I}$ is the intensity vector (the Stokes vector), s is the distance along the propagation path, $\mathbf{K}$ is the extinction matrix, $\mathbf{a}$ is the absorption vector and B is the source function (a scalar). If local thermodynamic equilibrium applies, B equals the Planck function describing blackbody radiation. See further Chapter 21.

As scattering here is neglected, the elements of $\mathbf{K}$ and $\mathbf{a}$ are linked to each other, and we have that[1]:

$$\mathbf{K}^{-1}\mathbf{a} = \begin{bmatrix} \alpha \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

where α is the total gas absorption coefficient.

[1] This equation is for sure valid for unpolarised absorption. We have not confirmed generally for polarised absorption, but appears to be valid for Zeeman splitting.

History

? Started by ?.

9.2 Standard algorithm

9.2.1 Simulation of transmission measurements

...

Chapter 10

Sensor modeling

A sensor model is needed because a practical instrument gives consistently spectra deviating from the hypothetical monochromatic pencil beam spectra provided by the atmospheric part of the forward model (that is $\mathbf{y} \neq \mathbf{i}$ always). For a radio (heterodyne) instrument, the most influential sensor parts are the antenna, the mixer, the sideband filter and the spectrometer. Limb sounding observations are also affected by Doppler shifts, but this effect is not considered here, it is assumed to be treated separately.

In the follow text we will use the terms sensor to denote the total sensor configuration, i.e. the the whole object that will be represented by the WSV `sensor_response`. To denote individual parts of the sensor, such as the antenna or the mixer, we use the terms sensor parts or instrument, especially when talking of characteristics.

10.1 Internal functions

As described in Section 4 the forward model is divided into an atmospheric part and a sensor part, Equation 4.6. The sensor part can be modelled by a sensor transfer matrix which is precalculated for the entire atmospheric simulation, and applied on the monochromatic pencil beam spectra, Equation 4.9. To calculate the total sensor response each individual part of the sensor has to be calculated and combined. The individual parts for a microwave measurements consists in most cases of an antenna, a sideband filter, a mixer and a spectrometer. These are the instruments that has been used as a starting point for the sensor modelling and WSM to compute their transfer matrices has been implemented. The description of the sensor response modelling is also found in *Eriksson et al.* [2006].

The operations performed by the instruments on the incoming spectra can either be described by a weighting, over viewing angles or frequencies, or a summation of signals in different sidebands. Summation and weighting of the spectral components are both linear operations, and thus it is possible to model the effect of the different sensor parts as subsequent matrix multiplications of the monochromatic pencil beam spectrum, as suggested in

History

050121 Revised and extended by Mattias Ekström.

000826 Written for ARTS-1 by Patrick Eriksson.

Here	In ARTS	Description
$w_{\rm a}$	`antenna_diagram`	The antenna pattern
$w_{\rm b}$		The backend channel response
$w_{\rm sb}$		The sideband filter function
ν_{ch}	`f_backend`	The frequency grid of the backend channels
$\nu_{\rm LO}$	`lo`	Local oscillator frequency
$\mathbf{P}$	`sensor_pol`	The sensor polarisation response
$\mathbf{H}$	`sensor_response`	The total sensor response matrix

Table 10.1: Examples of symbols used in this chapter, the corresponding notation in the ARTS source code and a short description of the quantity.

Eriksson et al. [2000]:

$$\mathbf{y} = \mathbf{H}_n \dots \mathbf{H}_2 \mathbf{H}_1 \mathbf{i} + \varepsilon \tag{10.1}$$

where n is the number of sensor parts to consider. Combining all sensor parts the sensor model can then be expressed as a single matrix multiplication (Eq. 4.9)

$$\mathbf{y} = \mathbf{H}\mathbf{i} + \varepsilon$$

Applying Equation 4.9 for the sensor model will clearly give very rapid calculations, and we must find ways to calculate $\mathbf{H}$. Therefore internal functions has been designed to calculate the response of such a weighting or a summation.

10.1.1 Weighting

The instruments that perform a weighting of the spectra are the antenna which weights different viewing angles by the normalised antenna response, $w_{\rm a}$, and the spectrometer that weights signals of neighbouring frequencies by the channel response, $w_{\rm ch}$. The influence of the antenna can be expressed as

$$i_{\rm a} = \int_{4\pi} i(\Omega + \Omega_0) w_{\rm a}(\Omega) {\rm d}\Omega \quad \text{with} \quad \int_{4\pi} w_{\rm a}(\Omega) {\rm d}\Omega = 1. \tag{10.2}$$

where i is the monochromatic pencil beam spectra, $i_{\rm a}$ the apparent radiation intensity after the antenna, Ω the solid angle and Ω_0 is some reference point for the antenna pattern, normally the point of highest response. For the spectrometeral the output for channel n is

$$\mathbf{y}_n = \int_0^\infty i(\nu) w_{\rm ch}^n(\nu) {\rm d}\nu \quad \text{with} \quad \int_0^\infty w_{\rm ch}^n(\nu) {\rm d}\nu = 1. \tag{10.3}$$

To incorporate this weighting into the response matrix $\mathbf{H}$, we want to express the weighting as a multiplication between two vectors:

$$\mathbf{hg} = \int f(x) g(x) {\rm d}x, \tag{10.4}$$

where x is either angle or frequency, $f(x)$ is the instrument response, $g(x)$ is the spectral signal, $\mathbf{g}$ is the vector representation of $g(x)$ and $\mathbf{h}$ is a row vector. The vector $\mathbf{h}$ corresponds to elements of a row in $\mathbf{H}$, where row and element positions depend on the considered response and sorting order between Stokes components, frequencies and viewing directions. It should be noted that the actual values of g are not known when creating $\mathbf{h}$, only the grid used for $\mathbf{g}$. In addition, the strength of the described method is that the same $\mathbf{H}$ can be used repeatedly, and then in conjunction with a varying $\mathbf{g}$.

The functions f and g can be represented using different grids, as indicated in Figure 10.1. This means that the product between f and g can have a discontinuity at each position corresponding to a grid point of either f or g. Letting k be the calculation interval index, the integral in Equation 10.4 is broken down as

$$\int_{x_k}^{x_{k+1}} f(x)g(x)\mathrm{d}x = \frac{x_{k+1}-x_k}{2}\Big(f(x_k)g(x_k) + f(x_{k+1})g(x_{k+1})\Big). \tag{10.5}$$

The value $g(x_k)$ is

$$g(x_k) = g(x_i)\frac{x_{i+1}-x_k}{x_{i+1}-x_i} + g(x_{i+1})\frac{x_k-x_i}{x_{i+1}-x_i}, \quad x_i \leq x_k < x_{i+1}, \tag{10.6}$$

where i is grid index of g, and the range $[x_i, x_{i+1}]$ encompasses the range $[x_k, x_{k+1}]$. The value $g(x_{k+1})$ is determined likewise. The response function f is known and $f(x_k)$ and $f(x_{k+1})$ can be explicitely calculated (following Equation 10.6).

Inserting Equation 10.6 in 10.5 yields the following expressions for the considered integral interval:

$$\begin{gathered}\textstyle\int_{x_k}^{x_{k+1}} f(x)g(x)\mathrm{d}x = \\ \tfrac{x_{k+1}-x_k}{2}\Big[\Big(f(x_k)\tfrac{x_{i+1}-x_k}{x_{i+1}-x_i} + f(x_{k+1})\tfrac{x_{i+1}-x_{k+1}}{x_{i+1}-x_i}\Big)g(x_i) + \\ \Big(f(x_k)\tfrac{x_k-x_i}{x_{i+1}-x_i} + f(x_{k+1})\tfrac{x_{k+1}-x_i}{x_{i+1}-x_i}\Big)g(x_{i+1})\Big],\end{gathered} \tag{10.7}$$

and the weights to be added to element i and $i+1$ of $\mathbf{h}$ are then

$$\mathbf{h}_i = \mathbf{h}_i + \tfrac{x_{k+1}-x_k}{2}\Big[f(x_k)\tfrac{x_{i+1}-x_k}{x_{i+1}-x_i} + f(x_{k+1})\tfrac{x_{i+1}-x_{k+1}}{x_{i+1}-x_i}\Big] \tag{10.8}$$

$$\mathbf{h}_{i+1} = \mathbf{h}_{i+1} + \tfrac{x_{k+1}-x_k}{2}\Big[f(x_k)\tfrac{x_k-x_i}{x_{i+1}-x_i} + f(x_{k+1})\tfrac{x_{k+1}-x_i}{x_{i+1}-x_i}\Big] \tag{10.9}$$

The vector $\mathbf{h}$ is initiated to hold zeros and the calculation procedure is iterated over k. To ensure energy conservation this vector should be normalised such that the sum of the elements equals one.

10.1.2 Summation

The summation takes place when the mixer downconverts the sideband signals into the intermediate frequency, ν_{IF}. This is done by mixing the measurement signal with the local oscillator signal (LO). For a frequency ν, the intermediate frequency is given by

$$\nu_{\mathrm{IF}} = |\nu - \nu_{\mathrm{LO}}|, \tag{10.10}$$

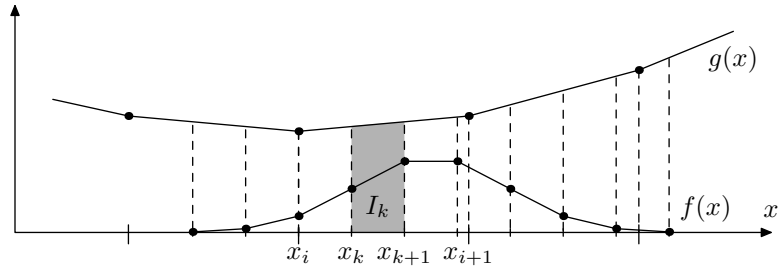

Figure 10.1: Schematic of antenna and backend calculations. The dashed lines show the end points of each integral interval. The shaded area corresponds to the interval considered in Equation 10.5.

where $\nu_{\rm LO}$ is the frequency of the LO. This procedure has the consequence that frequencies with the same distance to $\nu_{\rm LO}$ are translated to the same $\nu_{\rm IF}$. Thus, the 'image' frequency, ν', corresponding to ν is

$$\nu' = 2\nu_{\rm LO} - \nu. \tag{10.11}$$

The signal from ν and ν' is normally not weighted equally. This can be an effect of a frequency dependent response of elements from the antenna to the mixer, or caused by a sideband filter, with the purpose of minimising the contribution from one of the bands. These terms are here lumped together into one term, the sideband response $w_{\rm sb}$. The apparent intensity after the mixer, $I_{\rm IF}$, can then be expressed as

$$I_{\rm IF}(\nu_{\rm IF}) = \frac{w_{\rm sb}(\nu)I(\nu) + w_{\rm sb}(\nu')I(\nu')}{w_{\rm sb}(\nu) + w_{\rm sb}(\nu')}. \tag{10.12}$$

To model the sideband folding, a vector $\mathbf{h}$ representing the summation of the sideband filtered radio frequency (RF) signals into an intermediate frequency (IF) signal is to be determined. The translation of frequencies is given by Equation 10.11. To preserve all spectral information it is needed that the IF grid includes all unique transformations from RF to IF. That is, the IF grid shall be constructed by the projection of the RF grid from both sidebands (with duplicates of IF values removed). This implies that each IF grid point corresponds directly to a RF of one of the sidebands, but will normally fall between RF grid points in the other band, as illustrated in Figure 10.2, if not the RF grid is perfectly symmetric around the LO frequency.

Using a vector representation $\mathbf{v}$ of the RF grid and let $\mathbf{v}_i$ and $\mathbf{v}_{i+1}$ encompass the frequency ν and $\mathbf{v}_j$ and $\mathbf{v}_{j+1}$ encompasses ν', then the same notation as in Equation 10.12 can be used to write the elements of the response vector $\mathbf{h}$ as

$$\mathbf{h}_i = \frac{\mathbf{v}_{i+1} - \nu}{\mathbf{v}_{i+1} - \mathbf{v}_i} \frac{w_{\rm sb}(\nu)}{w_{\rm sb}(\nu) + w_{\rm sb}(\nu')}, \tag{10.13}$$

$$\mathbf{h}_{i+1} = \frac{\nu - \mathbf{v}_i}{\mathbf{v}_{i+1} - \mathbf{v}_i} \frac{w_{\rm sb}(\nu)}{w_{\rm sb}(\nu) + w_{\rm sb}(\nu')}, \tag{10.14}$$

$$\mathbf{h}_j = \frac{\mathbf{v}_{j+1} - \nu'}{\mathbf{v}_{j+1} - \mathbf{v}_j} \frac{w_{\rm sb}(\nu')}{w_{\rm sb}(\nu) + w_{\rm sb}(\nu')}, \tag{10.15}$$

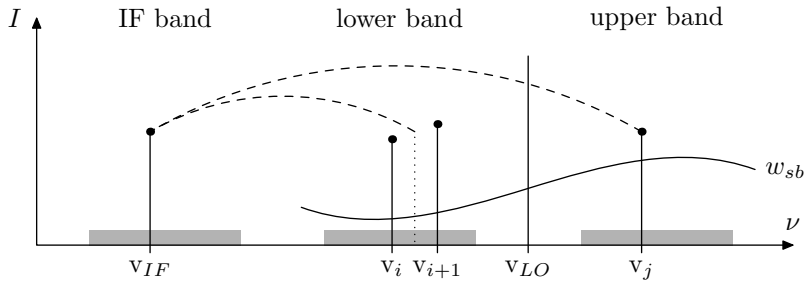

Figure 10.2: Visualisation of how the RF sidebands are folded around $\nu_{\rm LO}$ down to the IF. The sideband filter function, $w_{\rm sb}$, shows how the sidebands are weighted.

$$\mathbf{h}_{j+1} \quad = \quad \frac{\nu' - \mathbf{v}_j}{\mathbf{v}_{j+1} - \mathbf{v}_j} \frac{w_{\rm sb}(\nu')}{w_{\rm sb}(\nu) + w_{\rm sb}(\nu')}, \tag{10.16}$$

where all other elements of $\mathbf{h}$ are zero. Either $\mathbf{h}_{i+1}$ or $\mathbf{h}_{j+1}$ is unnecessary to compute, depending on if the IF corresponds directly to $\mathbf{h}_i$ or $\mathbf{h}_j$. The complete response matrix is computed by iterating over all IF.

The normal situation should be that the sideband folding and filtering are kept constant, and it is not needed to repeat the calculations even if the instrument is scanning. The weights of $\mathbf{h}$ from one viewing direction can for such cases be applied throughout the scan.

10.2 Instrument characteristics

To be able of computing the response from a certain sensor part, we need the instrument characteristics. This information thus have to be stored in such a way that we can incorporate it in our forward model. ARTS uses XML-files to store and retrieve variables from files and this offer a easy way to include characteristics that has been prepared outside ARTS. There also exist methods for computing simple gaussian characteristics. See Section 10.2.1 below for more details.

To describe sensor characteristics in ARTS matrices are often used with the first column containing relative grid positions for the values in the following columns. These grid positions can for example be frequencies relative to a channel frequency or angles relative to a antenna line-of-sight. These matrices can then be grouped together in arrays, or even array of arrays, to enable the characteristics to vary with other parameters such as polarisation, and use of different antennas. Common to all these variables are that the user has the choice that for each parameter either provide one set of characteristics that will be used for all cases or to provide a set of characteristics for each state of the parameter. As an example, consider the array of matrices used for the backend response characteristics. On the matrix level, the first column holds the relative frequency grid and the following hold the response values. If we want the same response for all channel frequencies only one column of response values is needed, on the other hand if we want the backend response to vary with the channel frequencies a column must be given for each frequency in ν_b. Further on the array level, if the array only has one matrix that response will be used for all polarisations otherwise the array needs as many matrices as polarisations simulated. Note that polarisations is not

the same as Stokes dimension, see Section 10.5 for more information on polarisation. How the sensor characteristics are implemented for each sensor part are described in the sections concerning the different sensor parts and in the online help.

10.2.1 Gaussian response

There exist a simple method for creating a gaussian instrument characteristic. Given the FWHM and the total width and maximum spacing of the grid `GaussianResponse` creates a matrix where the first column holds the relative grid and the second holds the weights. This method has a limited use, but can prove useful for setting up the characteristics for simple sensor configurations.

10.2.2 Normalisation

In most cases the sensor characteristics should be normalised as in Equation 10.2, 10.3 and 10.12, but for the possibility to study the sensor response, the normalisation is controlled by the WSV `sensor_norm`. For normalised sensor responses this variable should be set to one, and for non-normalised sensor responses it should be zero.

10.3 Sensor response initialisation

Even if the monochromatic pencil beam spectrum is studied without any sensor influence, the sensor variables, especially $\mathbf{H}$ has to be set. The reason is that the radiative transfer calculations allways applies a $\mathbf{H}$ to the spectrum or spectra. On the other hand if a response from a sensor system should be applied to the spectral values the sensor variables has to be initialised properly. For these tasks there exist two WSMs, `sensorOff` and `sensor_responseInit`. At least one of these has to be included in the control file.

10.3.1 No sensor

If the calculated monochromatic pencil beams is to be studied as is, i.e. there is no sensor system present in the simulation, only one WSM is needed `sensor_off`. It sets all the necessary sensor variables so that the monochromatic pencil beams are unchanged. This means for instance setting $\mathbf{H}$ to be an identity matrix. This method also affects several other workspace variables, to get a complete list use the online help for `sensor_off`. The syntax to disable the sensor in the simulation is simply;

```
sensorOff{}
```

10.3.2 Initialisation

When there is a sensor system present in the simulation the sensor response variables has to be set up in a correct way. The WSM `sensor_responseInit` takes care of this. It sets the $\mathbf{H}$ to be an identity matrix with correct size so that it is applicable to the monochromatic pencil beam column vector. It also initialises WSV that are use to remember the current settings of the spectra, these are updated by all succeeding sensor response WSMs, and

therefor gives the accurate output grids and values for the frequency and zenith- and azimuth angle grids and number of polarisations for the final spectra. These variables are

`sensor_response_f`	the frequency grid
`sensor_response_za`	the zenith angle grid
`sensor_response_aa`	the azimuthal angle grid
`sensor_response_pol`	the number of polarisations.

For the initial sensor response the output grids are the frequency grid of the monochromatic pencil beam calculations, the zenith- and azimuthal angle grid for each measurement block and the number of polarisation equal the Stokes dimension. Since these WSV are used for the initialisation they have to be set prior to calling `sensor_responseInit` together with `sensor_norm`, the position and line-of-sight of the sensor and the atmosphere dimension.

10.4 Antenna response

The influence of the antenna can as described above in Equation 10.2 be expressed as a weighting between the antenna response, w_{a}, as a function of viewing direction and the monochromatic pencil beam spectra. In ARTS the pencil beams are calculated for each measurement block at the angles given by zenith- and azimuthal angle grids added to the current line-of-sight as described in 3.4.3. Which viewing directions that are taken into account in the weighting is determined by the antenna line-of-sight and the antenna pattern. ARTS is prepared to handle a full 3D environment with a 2D antenna, but so far only methods for 1D antennas have been developped.

10.4.1 Antenna diagram

The antenna pattern w_{a} in ARTS is represented by a structure `antenna_diagram` that consists of an array of arrays of matrices. The reason to use such a complicated structure is to enable as much variability as possible when describing the antenna pattern. With this structure it is possible to define the antenna response for different viewing angles, polarisations and frequencies. Since there only exists a method for 1D antennas, w_{a} is so far only defined for 1D antenna patterns. The structure consists of matrices that describe the weights for different angles. The first column in matrices describes the relative angular grid and the following columns the weights for different frequencies, see Figure 10.4.1. Combining the antenna patterns relative angular grid with the antenna line-of-sight relative angle gives the measurement block angular grid seen by the antenna. Each polarisation is then represented by a separate matrix and each viewing angle by an array of matrices. By using arrays of matrices instead of tensors it is also possible to use different settings for the different polarisations/viewing directions. For each of these sets it is possible to define only one column, matrix or array of matrices, or a full set of columns, matrices or arrays of matrices. E.g. for the most simple case where the antenna has the same pattern for all directions and frequencies w_{a} consists of a single array of matrix with a single two column matrix.

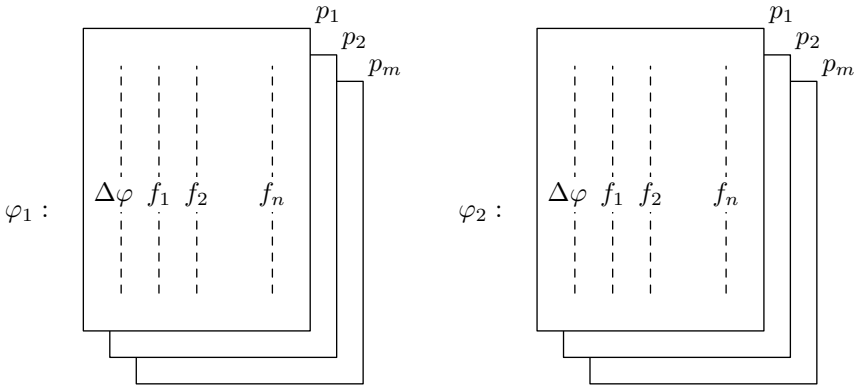

Figure 10.3: Visualisation of the antenna pattern WSV `antenna_diagram` for two viewing directions φ_1 and φ_2, $\Delta\varphi$ is the relative angular grid, f_i the i:th frequency and p_j the j:th polarisation.

10.4.2 Antenna line-of-sight

As described in 10.4.1 the antenna pattern includes a dimension for different viewing directions. This is useful when simulating multiple antennas or a multiple beam antenna. The viewing directions are described in ARTS by `antenna_los`. This is a one- or two column matrix describing the relative offset of the antennas with respect to the sensor line-of-sight. The columns follow the general idea for antenna dimensionality, with first column denoting zenith offsets and the second being used for 2D antennas and denoting azimuthal offsets. The rows on the other hand describes the number of antennas or beams, and in the common case of only one antenna present the matrix should therefor contain a row of zero. This variable is important when dealing with rotating sensors as is described in Section 10.5.2.

10.4.3 1D antenna

Here it will be assumed that the variation of I in one angular dimension can be neglected, and that the weighting can be described by an one dimensional integral:

$$I_{\mathrm{a}} = \int_{-\pi}^{\pi} I(\theta+\theta_0) w_{\mathrm{a}}^{\theta}(\theta)\,\mathrm{d}\theta, \tag{10.17}$$

$$w_{\mathrm{a}}^{\theta}(\theta) = \int_{0}^{2\pi} w_{\mathrm{a}}(\theta,\phi)\cos(\theta)\,\mathrm{d}\phi, \tag{10.18}$$

where θ_0 is the reference angle of the antenna, the weighting is only performed over the zenith angles. The only variables needed apart from the variables needed for the initialisation are the antenna line-of-sight and the antenna pattern. A convenient way to set the antenna dimensionality before the sensor initialisation is to use the WSM `AntennaSet1D`.

The 1D antenna is included in the sensor by using the WSM `sensor_responseAntenna1D` and it updates both $\mathbf{H}$ and the zenith angular grid of the final spectra.

10.5 Polarisation and rotation

Taking polarisation into account, and assuming no losses, the measured intensity for a certain direction and frequency , I_p, is

$$I_\mathrm{p} = \frac{1}{2}\mathbf{ps} \tag{10.19}$$

where $\mathbf{s}$ is the Stokes components of the incoming radiation and $\mathbf{p}$ is a row vector of the same length as $\mathbf{s}$ describing the sensor polarisation response. For a rotating sensor a transformation matrix $\mathbf{L}(\chi)$, see Section 10.5.2, is applied to obtain consistent definition between the polarisation directions for atmospheric radiation and sensor response, Equation 10.19 then becomes

$$I = \frac{1}{2}\mathbf{pL}(\chi)\mathbf{s}. \tag{10.20}$$

For an instrument measuring a single polarisation, the first element of $\mathbf{p}$ shall be one and the three last elements shall fulfil $\sum_{i=2}^{4} p_i^2 = 1$, where p_i is the i:th element of $\mathbf{p}$. For example, if vertical polarisation is measured, then $\mathbf{p} = [1\ 1\ 0\ 0]$ and for linear $\pm 45°$ polarisation $\mathbf{p} = [1\ 0\ \mp 1\ 0]$.

Several sensor parts can have a polarisation varying response, but there is normally a single part that dominates the polarisation response. The normal case for the instrument type considered here is that the mixer is only sensitive to a single polarisation, and the natural time to apply Equation 10.19 or 10.20 is then after the antenna, but before the sideband folding and the backend.

10.5.1 Polarisation response

To study different polarisations in ARTS, these has to be defined by the polarisation matrix $\mathbf{P}$. This is a matrix where each row is a row vector $\mathbf{p}$ corresponding to the polarisation to study and the columns matches the Stokes components of the simulation. The number of columns must therefor equal the Stokes dimension of the simulation, see Section 22 for more information and especially Section 22.3 for examples of some standard polarisations. An example matrix for studying horisontally and right-handed circular polarisation would look like

$$\mathbf{P} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}.$$

When initialised $\mathbf{P}$ is set to be the identity matrix, so to study all Stokes components nothing more has to be done. The `sensor_pol` matrix is most easily read from XML-file, and the calculation of the sensor response of polarisation is done by calling `sensor_responsePolarisation`. After this method the number of polarisations are equal to the rows of $\mathbf{P}$. When applying a polarisation matrix to the monochromatic pencil beam spectra the intensities are no longer ordered by Stokes dimension, instead they get ordered by the polarisations given in $\mathbf{P}$.

10.5.2 Rotating sensor

A rotating sensor configuration can be modelled by rotating the sensor frame between each measurement block by an angle χ. The transformation matrix for an angle χ is given by *Liou* [2002],

$$\mathbf{L}(\chi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 2\chi & \sin 2\chi & 0 \\ 0 & -\sin 2\chi & \cos 2\chi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{10.21}$$

In ARTS the rotation angles χ is stored in the vector `sensor_rot`, and is used to assign a rotation angle for each antenna line-of-sight. If the rotation is constant the `sensor_rot` vector can contain a single element describing that rotation angle. From Equation 10.20 it can be seen that the rotation should be applied to the incoming intensities before the polarisation, therefor the call to `sensor_responseRotation` must precede the call to the polarisation WSM in the control file. Since the rotation WSM does not know anything about viewing directions it must also be placed after the antenna function.

10.6 Mixer and sideband filter response

As described in Section 10.1.2 the mixer folds the two RF sidebands together into the IF band. At the same time the sideband ratio can be altered by the mixer itself and a sideband filter, these two effects are grouped together to what we refer to as the sideband filter. In ARTS there are methods both for a single mixer configuration and for multiple mixer configurations. From Equation 10.12 it can be seen that the summation is by its definition normalised, so even if `sensor_norm` is set to one the rows of $\mathbf{H}_{\text{mixer}}$ will be normalised.

10.6.1 Single mixer and sideband filter

For the single mixer configuration the needed variables are the LO frequency, ν_{LO}, and the sideband filter function, w_{sb}. In ARTS the ν_{LO} is stored in the vector `lo` which in this case should have length one, and w_{sb} is given as a matrix, the first column holding the RF grid and the second column the weights. There are no special WSV dedicated for the sideband filter, so it needs to be loaded into the arbitrary matrix variable `matrix_1` before being passed as a generic input to the calculating routine. This could look like

```
ReadXML( lo ) { "lo.xml" }
ReadXML( matrix_1 ) { "sbfilter.xml" }
sensor_responseMixer( matrix_1 ) {}
```

The method `sensor_responseMixer` changes the sensor response grid for frequencies to be in IF, this grid is also stored in `f_mixer`. Depending on the RF grid configuration, the conservation of all spectral values in RF may increase the number of grid points in IF.

10.6.2 Multiple mixers with single backends

For a sensor configuration with several channels which are related to different mixers a special WSM have been designed `sensor_responseMultiMixerBackend`. This

method works a bit different from the individual mixer and backend WSM. In this method each channel is related to a polarisation state, a local oscillator and a backend frequency channel. These three variables have to be declared for each channel, even if they are equal for some of the channels. The WSV that hold these variables are `sensor_pol`, `lo` and `f_backend`, and these should all have the same number of rows/length. For each channel the specified polarisation state is constructed from the Stokes vector, and a ν_{LO} is specified folding the spectra as in Equation 10.12. The w_{sb} can be given different weights for the different channels, these have to be given for the same relative grid and should be stored as subsequent columns in the generic input matrix. This method also applies the backend response to the spectra, this means a weighting over frequencies see Equation 10.3, with the possibility of using different backend responses for the different channels. Formally we can write the resulting sensor response as the product of the mixer and backend responses,

$$\mathbf{H}_{\mathrm{out}} = \mathbf{H}_{\mathrm{backend}}\mathbf{H}_{\mathrm{mixer}} \tag{10.22}$$

This implies that the backend channel characteristics, w_{b}, and the channel frequencies, ν_{ch}, has to be stored in their WSV prior to calling the WSM. This method operates in RF therefore ν_{ch} also has to be given in that domain, this also set the new sensor response frequency grid to be in RF. An important difference to the `sensor_responseBackend` method is that the backend response is given as a single matrix instead of an array of matrices. For more information about the backend response see Section 10.7.

10.6.3 Conversion of IF to RF

Since the mixer methods change the sensor response grid into IF, it can sometimes be convenient to be able to convert them back into the RF domain for a more understandable result. This is also a necessary step to perform if the radiances are to be converted into brightness temperatures. The conversion is done by the method `ConvertIFToRF` that given the generic input for which sideband to return, changes the sensor response frequency grid and if necessary re-orders the spectral values. If the IF grid is unfolded to double sidebands the number of grid points in RF will be twice as many as in IF, thus increasing the size of the sensor response frequency vector and the measurement vector. This method only works for single mixer configurations where there is only one ν_{LO} given.

10.7 Backend response

The last sensor part in the instrument is generally called the backend, sometimes the mixer is also included in this concept but here we only consider the part that transform monochromatic radiances into channel frequency radiances. In the case of a radiometer, this can be a spectrometer. Like the antenna weights the monochromatic pencil beams over angles, the backend spectrometer performs a similar weighting over frequencies as described by Equation 10.3. The spectra goes from being monochromatic to include intensities from the neighbouring frequencies. In ARTS the weighting is controlled by two variables; the channel centre frequencies ν_{ch} and the spectrometer channel response w_{ch}. The channel centre frequencies are the frequencies for which the spectrometer outputs the measured radiances, for this purpose the WSV `f_backend` is a vector that has to be provided by the user. The spectrometer channel response describes how the monochromatic radiances are

weighted around the centre frequencies, in analogy with the antenna line-of-sight and antenna pattern. The channel response is stored as an array of matrices, this way the weights are allowed to vary with frequency and polarisation. The matrices consists of a relative frequency grid and subsequent weights. Each matrix corresponds to the polarisations given in $\mathbf{P}$. The $w_{\rm ch}$ array of matrices can be passed to `sensor_responseBackend` by the generic input `arrayofmatrix_1`. This WSV can be set up by reading XML-files or by calling the function `ArrayOfMatrixSet`, see the online help for instructions. After applying the sensor response for the backend spectrometer the sensor response frequency grid is updated to be equal the $\nu_{\rm ch}$ grid.

10.8 Control file example

Chapter 11

Batch calculations

In many occasions you want to repeat the calculations with only a few variables changed. Examples on such cases are to perform 1D calculations for a number of atmospheric states taken from some atmospheric model, generate a set of spectra to create a training database for regression based inversions or perform a numeric inversion error analysis. For such calculations it is inefficient to perform the calculations by calling ARTS repeatedly. For example, as data must be imported for each call even if the data are identical between the cases. Cases such as the ones described above are here denoted commonly as batch calculations.

It is impossible to put all possible types of batch calculations inside a simple framework and the task of organising, or creating, the input data is left to more flexible programming environments, such as Matlab (where you of course use Atmlab) and Python. Instead, a general core functionality has been created that is useful for a large range of tasks.

11.1 Workspace variables and methods

The calculation flow is controlled be setting the `ybatch_calc_agenda`. Beside this agenda, the user must set the variable `ybatch_n`. This variable is the number of defined batch cases. Or in fact, how many batch cases that shall be considered. If the agenda can provide more than `ybatch_n` cases, the remaining cases are just ignored. There are few formal requirements on the involved agenda. Execution of `ybatch_calc_agenda` must result in a new spectrum vector, `y`, most likely by a call of `RteCalc`.

The actual calculations are made by calling `ybatchCalc`. The basic idea is simple, each loop inside this method shall produce a new spectrum vector, where each realisation is stored in `ybatch`. Variables are updated to a new batch case by `ybatch_calc_agenda`, where `ybatch_index`, set by `ybatchCalc`, tells the agenda which case to select. More in detail, `ybatchCalc` makes the following operations:

History	
070726	Copy-edited by Stefan Buehler.
070618	Updated by Oliver Lemke.
040916	Created by Patrick Eriksson.

1. Performs a-c with ybatch_index looped from 0 to (ybatch_n-1)

   a. Executes ybatch_calc_agenda.
   b. If ybatch_index = 0, resizes ybatch based on ybatch_n and length of y
   c. Makes copy of y in column ybatch_index of ybatch.

You see it is simple! The question is only what to put inside the agenda?

11.2 Control file examples

There exist some special workspace methods for the ybatch_calc_agenda, named as XxxxxExtractFromXxxxx. The common idea of these functions is to store the batch cases in tensors with one dimension extra compared to corresponding workspace variables. For example, a set of t_field (which is of type Tensor3) is stored as Tensor4.

In the following 1D example, five atmospheric scenarios have been put into the three first loaded files, and a random vector of zenith angles is found in the last file. The batch calculations are then performed as:

```
ReadXML( tensor4_1, "batch_t_field.xml" )
ReadXML( tensor4_2, "batch_z_field.xml" )
ReadXML( tensor5_1, "batch_vmr_field.xml" )
ReadXML( tensor3_1, "batch_za.xml" )
IndexSet( ybatch_n, 5 )

AgendaSet ( ybatch_calc_agenda ){
  Print( ybatch_index, 0 )
  Tensor3ExtractFromTensor4( t_field,    tensor4_1, ybatch_index )
  Tensor3ExtractFromTensor4( z_field,    tensor4_2, ybatch_index )
  Tensor4ExtractFromTensor5( vmr_field,  tensor5_1, ybatch_index )
  MatrixExtractFromTensor3(  sensor_los, tensor3_1, ybatch_index )

  RteCalc
}

ybatchCalc
MatrixToTbByRJ( ybatch, ybatch )

WriteXML( "ascii", ybatch, "ybatch_run1.xml" )
```

If you then want to repeat the calculations, for example with another propagation path step length (e.g. 25 km), it is sufficient to add the lines:

```
AgendaSet( ppath_step_agenda ){
   ppath_stepGeometric( ppath_step, atmosphere_dim, p_grid,
                        lat_grid, lon_grid,
                        z_field, r_geoid, z_surface,
```

```
                          25e3 )
}

ybatchCalc
MatrixToTbByRJ( ybatch, ybatch )

WriteXML( "ascii", ybatch, "ybatch_run2.xml" )
```

Chapter 12

Description of clouds

12.1 Introduction

In the Earth's atmosphere we find liquid water clouds consisting of approximately spherical water droplets and cirrus clouds consisting of ice particles of diverse shapes and sizes. We also find different kinds of aerosols. In order to take into account this variety, the model allows to define several *particle types*. A particle type is either a specified particle or a specified particle distribution, for example a particle ensemble following a gamma size distribution. The particles can be completely randomly oriented, azimuthally randomly oriented or arbitrarily oriented. For each particle type being a part of the modeled cloud field, a data file containing the single scattering properties ($\langle \mathbf{K}_i \rangle$, $\langle \mathbf{a}_i \rangle$, and $\langle \mathbf{Z}_i \rangle$), and the appropriate particle number density field is required. The particle number density fields are stored as `GField3`, including the field stored in a three-dimensional tensor and also the appropriate atmospheric grids (pressure, latitude and longitude grid). For each grid point in the cloud box the single scattering properties are averaged using the particle number density fields. In the scattering database the single scattering properties are not always stored in the same coordinate system. For instance for randomly oriented particles it makes sense to store the single scattering properties in the so-called scattering frame in order to reduce memory requirements. The following section describes in detail the `SingleScatteringData` class. The consequent section describes how to realize different kinds of size distributions in the ARTS frame by defining appropriate particle number density fields.

12.2 Single scattering properties

12.2.1 Coordinate systems: The laboratory frame and the scattering frame

For radiative transfer calculations we need a coordinate system to describe the direction of propagation. For this purpose we use the laboratory frame, which has been introduced in Section 3 and which is also shown in Figure 21.1. The z-axis corresponds to the local zenith direction and the x-axis points towards the north-pole. The propagation direction

History
050913 Created and written by Claudia Emde

is described by the local zenith angle θ and the local azimuth angle ϕ. This coordinate system is the most appropriate frame to describe the propagation direction and the polarization state of the radiation. However, in order to describe scattering of radiation by a particle or a particle ensemble, it makes sense to define another coordinate system taking into consideration the symmetries of the particle or the scattering medium, as one gets much simpler expressions for the single scattering properties. For macroscopically isotropic and mirror-symmetric scattering media it is convenient to use the scattering frame, in which the incidence direction is parallel to the z-axis and the x-axis coincides with the scattering plane, that is, the plane through the unit vectors $\hat{\mathbf{n}}^{\mathrm{inc}}$ and $\hat{\mathbf{n}}^{\mathrm{sca}}$. The scattering frame is illustrated in Figure 12.1. For symmetry reasons the single scattering properties defined with respect to the scattering frame can only depend on the scattering angle Θ,

$$\Theta = \arccos(\hat{\mathbf{n}}^{\mathrm{inc}} \cdot \hat{\mathbf{n}}^{\mathrm{sca}}), \tag{12.1}$$

between the incident and the scattering direction.

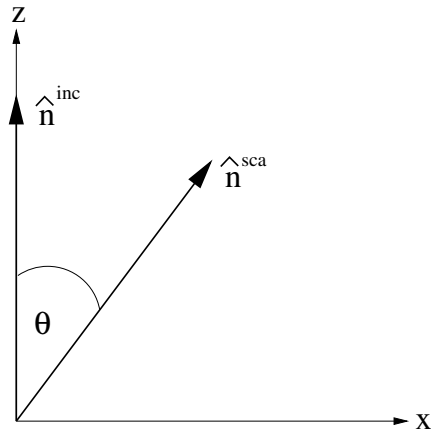

Figure 12.1: Illustration of the scattering frame. The z-axis coincides with the incident direction $\hat{\mathbf{n}}^{\mathrm{inc}}$. The scattering angle Θ is the angle between $\hat{\mathbf{n}}^{\mathrm{inc}}$ and $\hat{\mathbf{n}}^{\mathrm{sca}}$.

12.2.2 Scattering datafile structure

The single scattering properties are pre-calculated, for example by using the T-matrix code by *Mishchenko et al.* [2002], and stored in data-files. Different methods for the calculation of single scattering properties are reviewed in *Emde* [2005].

The format of the scattering database allows space reduction due to symmetry for certain special cases, e.g. random orientation or horizontal alignment. The file format is XML. The data is stored in a class called `SingleScatteringData`, which resides in the files `optproperties.h`. The class consists of the following fields (compare also Table 12.2.2):

- *enum* `ptype`: An attribute which contains information about the data type, which is the classification of the kind of hydrometeor species (randomly oriented, general case ...). This attribute is needed in the radiative transfer function to be able to extract

the physical phase matrix, the physical extinction matrix and the physical absorption vector from the data.

Possible values of ptype are:

`PTYPE_GENERAL` = 10
`PTYPE_MACROS_ISO` = 20
`PTYPE_HORIZ_AL` = 30

A more detailed description of the different cases is given below.

- *String* `description`: Here the particle type should be specified explicitly. We can have the case randomly oriented particles, but furthermore we also have to specify the exact particle properties (i.e. size and shape distribution). This can be a longer text describing how the scattering properties were generated. It should be formatted for direct printout to screen or file.
- *Vector* `f_grid`: Frequency grid [Unit: Hz].
- *Vector* `T_grid`: Temperature grid [Unit: K].
- *Vector* `za_grid`:
  1. `p10, p30`: Zenith angle grid (Range: $0.0° \leq$ za $\leq 180.0°$).
  2. `p20`: Scattering angle grid (Range: $0.0° \leq$ za $\leq 180.0°$).
- *Vector* `aa_grid`: Azimuth angle grid.
  1. `p10`: Range: $-180.0° \leq$ aa $\leq 180.0°$
  2. `p20`: Not needed, since optical properties depend only on scattering angle (dummy grid).
  3. `p30`: Only half of the grid is required (Range: $0.0° \leq$ aa $\leq 180.0°$)

  The angular grids have to satisfy the following conditions:
  - They have to be equidistant.
  - The value of the data must be the same for the first and the last grid-point. This condition is required for the integration routine.
  - If we only have to store a part of the grid, for example `za_grid` only from 0°to 90°, these two values (0°, 90°) must be grid-points.
- *Tensor7* `pha_mat_data`: Phase matrix data $\langle \mathbf{Z} \rangle$ [Unit: m^2]. The dimensions of the data array are:

  ```
  [frequency temperature za_sca aa_sca za_inc aa_inc
  matrix_element]
  ```

  The order of matrix elements depends on the chosen case. For most cases we do not need all matrix elements (see description of cases below).

Table 12.1: Structure of single scattering data files

Symbol	Type	Dimensions	Description
	enum		specification of particle type
	String		short description of particle type
ν	Vector	(ν)	frequency grid
T	Vector	(T)	temperature grid
ψ	Vector	(ψ)	zenith angle grid
ω	Vector	(ω)	azimuth angle grid
$\langle \mathbf{Z} \rangle$	Tensor7	$(\nu, T, \psi, \omega, \psi', \omega', i\ )$	phase matrix
$\langle \mathbf{K} \rangle$	Tensor5	$(\nu, T, \psi, \omega, i\ )$	extinction matrix
$\langle \mathbf{a} \rangle$	Tensor5	$(\nu, T, \psi, \omega, i\ )$	absorption vector

- *Tensor5* `ext_mat_data`: Extinction matrix data $\langle \mathbf{K} \rangle$ [Unit: m^2]. The dimensions are:

  `[frequency temperature za_inc aa_inc matrix_element]`

  Again, the order of matrix elements depends on the chosen case.

- *Tensor5* `abs_vec_data`: Absorption vector data $\langle \mathbf{a} \rangle$ [Unit: m^2].

  The absorption vector is also precalculated. It could be calculated from extinction matrix and phase matrix. But this calculation takes long computation time, as it requires an angular integration over the phase matrix. For the cases with symmetries (e.g., random orientation) the data files will not become too large even if we store additionally the absorption vector. The dimensions are:

  `[frequency temperature za_inc aa_inc vector_element]`

12.2.3 Definition of particle types

Macroscopically isotropic and mirror-symmetric scattering media (p20)

For macroscopically isotropic and mirror-symmetric scattering media (totally randomly oriented particles) the optical properties are calculated in the so-called scattering frame as shown in Figure 12.1. In this coordinate system the z-axis corresponds to the incident direction and the xz-plane coincides with the scattering plane. Using this frame only the scattering angle, which is the angle between incident and scattered direction is needed. Furthermore the number of matrix elements of both matrices, phase matrix and extinction matrix, can be reduced (see *Mishchenko et al.* [2002], p.90). To calculate the particle optical properties it is convenient to use Mishchenko's T-matrix code for randomly oriented particles [*Mishchenko and Travis*, 1998] which returns the averaged phase matrix and extinction matrix. The only drawback is that the single scattering data has to be transformed from the particle frame representation to the laboratory frame representation. These transformations are described in the appendix of *Emde* [2005].

Only six elements of the transformed phase matrix, which is commonly called scattering matrix **F**, are different. Therefore the size of `pha_mat_data` is:

```
[N_f N_T N_za_sca 1 1 1 6]
```

The order of the matrix elements is as follows: *F11, F12, F22, F33, F34, F44*
The extinction matrix is in this case diagonal and independent of direction and polarization. That means that we need to store only one element for each frequency. Hence the size of `ext_mat_data` is

```
[N_f N_T 1 1 1]
```

The absorption vector is also direction and polarization independent. Therefore the size of `abs_vec_data` for this case is the same as `ext_mat_data`:

```
[N_f N_T 1 1 1]
```

Horizontally aligned plates and columns (p30) For particle distributions of horizontally aligned plates and columns that are oriented randomly in the azimuth the angular dimension can be reduced by one, if we rotate the coordinate system appropriately. For this case we use the T-matrix code for single particles in fixed orientation and average phase matrix and extinction matrix manually like in the general case.

The phase matrix (and also extinction matrix and absorption vector) become independent of the incident azimuth angle in this frame. Furthermore, regarding the symmetry of this case, it can be shown that for the scattered directions we need only half of the angular grids, as the two halves must contain the same data. `pha_mat_data` therefore has the following size:

```
[N_f N_T N_za_sca N_aa_sca N_za_inc/2+1 1 16]
```

We store `za_sca` for all grid points from 0°to 180°, `aa_sca` from 0°to 180°, and `za_inc` from 0°to 90°. This means that the zenith angle grid has to include 90°as grid-point. The order of the matrix elements is the same as in the general case. For this case it can be shown that the extinction matrix has only three elements *Kjj*, *K12(=K21)*, and *K34(=-K43)*. Because of azimuthal symmetry, it can not depend on the azimuth angle. Hence the size of `ext_mat_data` is

```
[N_f N_T N_za/2+1 1 3]
```

The absorption coefficient vector has only two elements *a1* and *a2*. This means that the size of `abs_vec_data` is

```
[N_f N_T N_za/2+1 1 2]
```

General case (p10) If there are no symmetries at all we have to store all 16 elements of the phase matrix. The average phase matrix has to be generated from all individual phase matrices of the particles in the distribution outside ARTS. The individual phase matrices are calculated using Mishchenko's T-matrix code for single particles in fixed orientation [*Mishchenko*, 2000]. We have to store all elements for all angles in the grids. The size of `pha_mat_data` is therefore:

```
[N_f N_T N_za_sca N_aa_sca N_za_inc N_aa_inc 16]
```

The matrix elements have to be stored in the following order: *Z11, Z12, Z13, Z14, Z21, Z22, ...* Seven extinction matrix elements are independent (cp. *Mishchenko et al.* [2002], p.55). The elements being equal for single particles should still be the equal for a distribution as we get the total extinction just by adding. Here we need only the incoming grids, so the size of ext_mat_data is:

```
[N_f N_T N_za_inc N_aa_inc 7]
```

The absorption vector in general has four components (cp. Equation (2.186) in *Mishchenko et al.* [2002]). The size of abs_vec_data is accordingly:

```
[N_f N_T N_za_inc N_aa_inc 4]
```

Generating single scattering properties It is very convenient to use the PYTHON module PyARTS, which has been developed especially for ARTS and which is freely available at `http://www.sat.uni-bremen.de/cgi-bin/cvsweb.cgi/PyARTS/`. This module can be used to generate single scattering properties for horizontally aligned as well as for randomly oriented particles in the ARTS data-file-format. PyARTS has been developed by C. Davis, who has implemented the Monte Carlo scattering algorithm in ARTS (see Section 14). The ATMLAB package includes functions to generate single scattering properties for spherical particles (Mie-Theory).

12.3 Representation of the particle size distribution

The particle size has an important impact on the scattering and absorption properties of cloud particles as shown for instance in [*Emde et al.*, 2004]. Clouds contain a whole range of different particle sizes, which can be described by a size distribution giving the number of particles per unit volume per unit radius interval as a function of radius. It is most convenient to parameterize the size distribution by analytical functions, because in this case optical properties can be calculated much faster than for arbitrary size distributions. The T-matrix code for randomly oriented particles includes several types of analytical size distributions, e.g., the gamma distribution or the log-normal distribution. This section presents the size distribution parameterizations, which were used for the ARTS simulations included in this thesis.

12.3.1 Mono-disperse particle distribution

The most simple assumption is, that all particles in the cloud have the same size. In order to study scattering effects like polarization or the influence of particle shape, it makes sense to use this most simple assumption, because one can exclude effects resulting from the particle size distribution itself.

Along with the single scattering properties we need the particle number density field, which specifies the number of particles per cubic meter at each grid point, for ARTS scattering simulations. For a given IMC and mono-disperse particles the particle number density n^p is simply

$$n^p(IMC, r) = \frac{IMC}{m} = \frac{IMC}{V\rho} = \frac{3}{4\pi}\frac{IMC}{\rho r^3}, \tag{12.2}$$

where m is the mass of a particle, r is its equal volume sphere radius, ρ is its density, and V is its volume.

12.3.2 Gamma size distribution

A commonly used distribution for radiative transfer modeling in cirrus clouds is the *gamma distribution*

$$n(r) = ar^{\alpha}\exp(-br). \tag{12.3}$$

The dimensionless parameter α describes the width of the distribution. The other two parameters can be linked to the effective radius R_{eff} and the ice mass content IMC as follows:

$$b = \frac{\alpha+3}{R_{eff}}, \tag{12.4}$$

$$a = \frac{IMC}{4/3\pi\rho b^{-(\alpha+4)}\Gamma[\alpha+4]}, \tag{12.5}$$

where ρ is the density of the scattering medium and Γ is the gamma function. For cirrus clouds ρ corresponds to the bulk density of ice, which is approximately 917 kg/m^3.

Generally, the effective radius R_{eff} is defined as the average radius weighted by the particle cross-section

$$R_{eff} = \frac{1}{\langle A\rangle}\int_{r_{min}}^{r_{max}} A(r)rn(r)\mathrm{d}r, \tag{12.6}$$

where A is the area of the geometric projection of a particle. The minimal and maximal particle sizes in the distribution are given by r_{min} and r_{max} respectively. In the case of spherical particles $A = \pi r^2$. The average area of the geometric projection per particle $\langle A\rangle$ is given by

$$\langle A\rangle = \frac{\int_{r_{min}}^{r_{max}} A(r)n(r)\mathrm{d}r}{\int_{r_{min}}^{r_{max}} n(r)\mathrm{d}r}. \tag{12.7}$$

The question is how well a gamma distribution can represent the true particle size distribution in radiative transfer calculations. This question is investigated by *Evans et al.* [1998]. The authors come to the conclusion that a gamma distribution represents the distribution of realistic clouds quite well, provided that the parameters R_{eff}, IMC and α are chosen correctly. They show that setting $\alpha = 1$ and calculating only R_{eff} gave an agreement within 15% in 90% of the considered measurements obtained during the First ISCCP Regional Experiment (FIRE). Therefore, for all calculations including gamma size distributions for ice particles, $\alpha = 1$ was assumed.

The particle number density for size distributions is obtained by integration of the distribution function over all sizes:

$$n^p(IMC, R_{eff}) = \int_0^{\infty} n(r)\mathrm{d}r \tag{12.8}$$

$$= \int_0^{\infty} ar^{\alpha}\exp(-br)\mathrm{d}r = a\frac{\Gamma(\alpha+1)}{b^{\alpha+1}}. \tag{12.9}$$

After setting α = 1, inserting Equation (12.5) and some simple algebra we obtain

$$n^p(IMC, R_{eff}) = \frac{2}{\pi}\frac{IMC}{\rho R_{eff}^3}. \tag{12.10}$$

Comparing Equation (12.2) and (12.10), we see that the particle number density for monodisperse particles with a particle size of R is smaller than the particle number density for gamma distributed particles with $R_{eff} = R$. The reason is that in the gamma distribution most particles are smaller than R_{eff}.

12.3.3 Ice particle size parameterization by McFarquhar and Heymsfield

A more realistic parameterization of tropical cirrus ice crystal size distributions was derived by *McFarquhar and Heymsfield* [1997], who derived the size distribution as a function of temperature and IMC. The parameterization was made based on observations during the Central Equatorial Pacific Experiment (CEPEX). Smaller ice crystals with an equal volume sphere radius of less than 50 μm are parametrized as a sum of first-order gamma functions:

$$n(r) = \frac{12 \cdot IMC_{<50} \alpha_{<50}^5 r}{\pi \rho \Gamma(5)} \exp(-2\alpha_{<50} r), \tag{12.11}$$

where $\alpha_{<50}$ is a parameter of the distribution, and $IMC_{<50}$ is the mass of all crystals smaller than 50 μm in the observed size distribution. Large ice crystals are represented better by a log-normal function

$$n(r) = \frac{3 \cdot IMC_{>50}}{\pi^{3/2} \rho \sqrt{2} \exp(3\mu_{>50} + (9/2)\sigma_{>50}^2) r \sigma_{>50} r_0^3} \cdot \exp\left[-\frac{1}{2} \left(\frac{\log \frac{2r}{r_0} - \mu_{>50}}{\sigma_{>50}} \right)^2 \right], \tag{12.12}$$

where $IMC_{>50}$ is the mass of all ice crystals greater than 50 μm in the observed size distribution, r_0 = 1 μm is a parameter used to ensure that the equation does not depend on the choice of unit for r, $\sigma_{>50}$ is the geometric standard deviation of the distribution, and $\mu_{>50}$ is the location of the mode of the log-normal distribution. The fitted parameters of the distribution can be looked up in the article by *McFarquhar and Heymsfield* [1997]. The particle number density field is obtained by numerical integration over a discrete set of size bins. This parameterization of particle size has been implemented in the PyARTS package, which was introduced in Section 12.2.2. Using PyARTS one can calculate the size distributions, the corresponding single scattering properties and the particle number density fields for given IMC and temperature.

12.4 Implementation

The workspace methods related to the description of clouds in ARTS are implemented in the file `m_cloudbox.cc`. Work space methods related to the optical properties of the clouds are implemented in the file `m_optproperties.cc`. The coordinate system transformations described above reside in the file `optproperties.cc`.

12.4.1 Work space methods and variables

The following controlfile section illustrates how a simple cloud can be included in an ARTS calculation.

First we have to define the cloudbox region, i.e. the region where scattering objects are found. To do this we use the method `cloudboxSetManuallyAltitude`:

```
cloudboxSetManuallyAltitude( cloudbox_on, cloudbox_limits,
                             atmosphere_dim, p_grid,
                             lat_grid, lon_grid,
```

```
                                8000, 120000,
                                0, 0, 0, 0 )
```

If we want to do a simulation for a cirrus cloud at an altitude from 9 to 11 km the cloudbox limits can be set to 8 and 12 km. The latitude and longitude limits are set to an arbitrary value for a 1D calculation. For 3D calculations they are also needed. Alternatively one can use the method `cloudboxSetManually`, where one has to provide pressure instead of altitude limits.

Now we have to specify the cloud particles inside the scattering region:

```
# Initialisation
ParticleTypeInit
# Only one particle type is added in this example
ParticleTypeAdd( scat_data_raw, pnd_field_raw,
                 atmosphere_dim, f_grid, p_grid,
                 lat_grid, lon_grid, cloudbox_limits,
                 "ssd_sphere_50um_p20.xml",
                 "pnd_sphere_50um_p20.xml" )
```

In the workspace method `ParticleTypeAdd` the single scattering properties for one particle type are read. The generic input `filename_scat_data` must be set to the filename of a datafile including scattering data (class `SingleScatteringData`) in xml-format. The generic input `filename_pnd_field` must contain the filename of the corresponding particle number density field in xml-format (class `GField3`). If the cloud is composed of several different particle types `ParticleTypeAdd` can be used repeatedly for all particle types, for instance one could add to the randomly oriented spherical particles above horizontally aligned cylindrical particles:

```
ParticleTypeAdd( scat_data_raw, pnd_field_raw,
                 atmosphere_dim, f_grid, p_grid,
                 lat_grid, lon_grid, cloudbox_limits,
                 "ssd_cylinder_30um_p30.xml",
                 "pnd_cylinder_30um_p30.xml" )
```

Alternatively it is possible to use the method `ParticleTypeAddAll`, which is convenient to generate a size distribution using several size bins. In this case one needs to define one particle type for each size bin. For many size bins the control file becomes very lengthy if one uses `ParticleTypeAdd` repeatedly. `ParticleTypeAddAll` requires as input an array of string including all filenames of the single scattering data files and the variable `pnd_field_raw` which includes the particle number density fields for all particle types. Using this function, one has to make sure that the order of the filenames containing the single scattering data corresponds to the order of the particle number density fields in `pnd_field_raw`. After reading the data the workspace variable `pnd_field` is calculated using `pnd_fieldCalc`:

```
# Calculate the particle number density field
pnd_fieldCalc
```

The definition of the single scattering data along with the corresponding particle number density fields is common in both scattering modules, the DOIT module described in Chapter 13 and the Monte Carlo module described in Chapter 14.

Chapter 13

Scattering - DOIT module

The Discrete Ordinate ITerative (DOIT) method is one of the scattering algorithms in ARTS. Besides the DOIT method a backward Monte Carlo scheme has been implemented (see Section 14). The DOIT method is unique because a discrete ordinate iterative method is used to solve the scattering problem in a spherical atmosphere. Although the DOIT module is implemented for 1D and 3D atmospheres, it is strongly recommended to use it only for 1D, because the Monte Carlo module (Chapter 14) is much more appropriate for 3D calulations. More appropriate in the sense that it is much more efficient. A literature review about scattering models for the microwave region, which is presented in *Emde and Sreerekha* [2004], shows that former implementations of discrete ordinate schemes are only applicable for (1D-)plane-parallel or 3D-cartesian atmospheres. All of these algorithms can not be used for the simulation of limb radiances. A description of the DOIT method, similar to what is presented in this chapter, has been published in *Emde et al.* [2004] and in *Emde* [2005].

13.1 The discrete ordinate iterative method

13.1.1 Radiation field

The Stokes vector depends on the position in the cloud box and on the propagation direction specified by the zenith angle (ψ) and the azimuth angle (ω). All these dimensions are discretized inside the model; five numerical grids are required to represent the radiation field $\mathcal{I}$:

$$\begin{aligned} \vec{P} &= \{P_1, P_2, ..., P_{N_P}\}, \\ \vec{\alpha} &= \{\alpha_1, \alpha_2, ..., \alpha_{N_\alpha}\}, \\ \vec{\beta} &= \{\beta_1, \beta_2, ..., \beta_{N_\beta}\}, \\ \vec{\psi} &= \{\psi_1, \psi_2, ..., \psi_{N_\psi}\}, \end{aligned} \tag{13.1}$$

History	
020601	Created and written by Claudia Emde
050223	Rewritten by Claudia Emde, mostly taken from Chapter 4 of Claudia's PhD thesis
050929	Included technical part, example contol file

$$\vec{\omega} \quad = \{\omega_1, \omega_2, ..., \omega_{N_\omega}\}.$$

Here $\vec{P}$ is the pressure grid, $\vec{\alpha}$ is the latitude grid and $\vec{\beta}$ is the longitude grid. The radiation field is a set of Stokes vectors ($N_P \times N_\alpha \times N_\beta \times N_\psi \times N_\omega$ elements) for all combinations of positions and directions:

$$\begin{aligned}\mathcal{I} \quad &= \{\mathbf{I}_1(P_1, \alpha_1, \beta_1, \psi_1, \omega_1), \mathbf{I}_2(P_2, \alpha_1, \beta_1, \psi_1, \omega_1), ..., \\ &\quad \mathbf{I}_{N_P \times N_\alpha \times N_\beta \times N_\psi \times N_\omega}(P_{N_P}, \alpha_{N_\alpha}, \beta_{N_\beta}, \psi_{N_\psi}, \omega_{N_\omega})\}. \end{aligned} \tag{13.2}$$

In the following we will use the notation

$$\mathcal{I} = \{\mathbf{I}_{ijklm}\} \qquad \begin{matrix} i = 1 \ldots N_P \\ j = 1 \ldots N_\alpha \\ k = 1 \ldots N_\beta. \\ l = 1 \ldots N_\psi \\ m = 1 \ldots N_\omega \end{matrix} \tag{13.3}$$

13.1.2 Vector radiative transfer equation solution

Figure 13.1 shows a schematic of the iterative method, which is applied to solve the vector radiative transfer equation (21.35)

$$\frac{\mathrm{d}\mathbf{I}(\mathbf{n}, \nu, T)}{\mathrm{d}s} = \quad - \langle \mathbf{K}(\mathbf{n}, \nu, T) \rangle \, \mathbf{I}(\mathbf{n}, \nu, T) + \langle \mathbf{a}(\mathbf{n}, \nu, T) \rangle \, B(\nu, T) \tag{13.4}$$

$$+ \textstyle\int_{4\pi} \mathrm{d}\mathbf{n}' \, \langle \mathbf{Z}(\mathbf{n}, \mathbf{n}', \nu, T) \rangle \, \mathbf{I}(\mathbf{n}', \nu, T), \tag{13.5}$$

where $\mathbf{I}$ is the specific intensity vector, $\langle \mathbf{K} \rangle$ is the ensemble-averaged extinction matrix, $\langle \mathbf{a} \rangle$ is the ensemble-averaged absorption vector, B is the Planck function and $\langle \mathbf{Z} \rangle$ is the ensemble-averaged phase matrix. Furthermore ν is the frequency of the radiation, T is the temperature, $\mathrm{d}s$ is a path-length-element of the propagation path and $\mathbf{n}$ the propagation direction. Equation (21.35) is explained more detailed in Section 21.5.

The *first guess field*

$$\mathcal{I}^{(0)} = \left\{ \mathbf{I}^{(0)}_{ijklm} \right\}, \tag{13.6}$$

is partly determined by the boundary condition given by the radiation coming from the clear sky part of the atmosphere traveling into the cloud box. Inside the cloud box an arbitrary field can be chosen as a first guess. In order to minimize the number of iterations it should be as close as possible to the solution field.

The next step is to solve the scattering integrals

$$\left\langle \mathbf{S}^{(0)}_{ijklm} \right\rangle = \int_{4\pi} \mathrm{d}\mathbf{n}' \, \langle \mathbf{Z}_{ijklm} \rangle \, \mathbf{I}^{(0)}_{ijklm}, \tag{13.7}$$

using the first guess field, which is now stored in a variable reserved for the *old radiation field*. For the integration we use equidistant angular grids in order to save computation time (cf. Section 13.3.1). The radiation field, which is generally defined on finer angular grids ($\vec{\omega}, \vec{\psi}$), is interpolated on the equidistant angular grids. The integration is performed over all

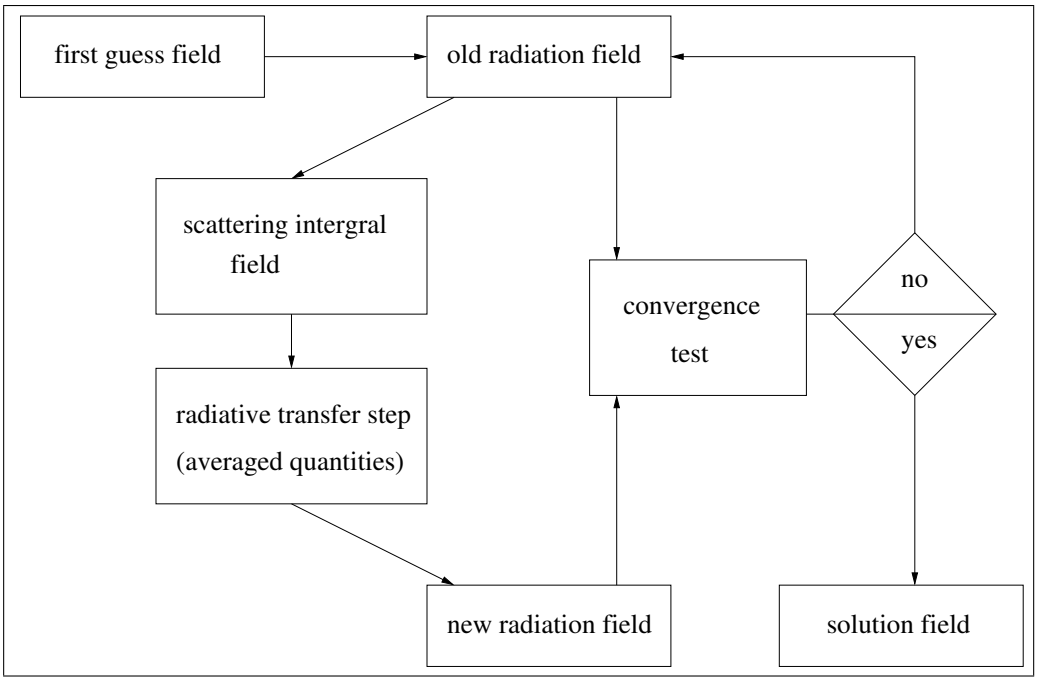

Figure 13.1: Schematic of the iterative method to solve the VRTE in the cloud box.

incident directions $\mathbf{n}'$ for each propagation direction $\mathbf{n}$. The evaluation of the scattering integral is done for all grid points inside the cloud box. The obtained integrals are interpolated on $\vec{\omega}$ and $\vec{\psi}$. The result is the first guess *scattering integral field* $\mathcal{S}^0$:

$$\mathcal{S}^{(0)} = \left\{ \left\langle \mathbf{S}^{(0)}_{ijklm} \right\rangle \right\}. \tag{13.8}$$

Figure 13.2 shows a propagation path step from a grid point $\mathbf{P} = (P_i, \alpha_j, \beta_k)$ into direction $\mathbf{n} = (\psi_l, \omega_m)$. The radiation arriving at $\mathbf{P}$ from the direction $\mathbf{n}'$ is obtained by solving the linear differential equation:

$$\frac{\mathrm{d}\mathbf{I}^{(1)}}{\mathrm{d}s} = -\overline{\langle \mathbf{K} \rangle} \mathbf{I}^{(1)} + \overline{\langle \mathbf{a} \rangle}\, \bar{B} + \overline{\langle \mathbf{S}^{(0)} \rangle}, \tag{13.9}$$

where $\overline{\langle \mathbf{K} \rangle}$, $\overline{\langle \mathbf{a} \rangle}$, $\bar{B}$ and $\overline{\langle \mathbf{S}^{(0)} \rangle}$ are *averaged quantities*. This equation can be solved analytically for constant coefficients. Multi-linear interpolation gives the quantities $\mathbf{K}', \mathbf{a}', \mathbf{S}'$ and T' at the intersection point $\mathbf{P}'$. To calculate the radiative transfer from $\mathbf{P}'$ towards $\mathbf{P}$ all quantities are approximated by taking the averages between the values at $\mathbf{P}'$ and $\mathbf{P}$. The average value of the temperature is used to get the averaged Planck function $\bar{B}$.

The solution of Equation (13.9) is found analytically using a matrix exponential approach:

$$\mathbf{I}^{(1)} = e^{-\overline{\langle \mathbf{K} \rangle} s} \mathbf{I}^{(0)} + \left(\mathbf{I} - e^{-\overline{\langle \mathbf{K} \rangle} s} \right) \overline{\langle \mathbf{K} \rangle}^{-1} \left(\overline{\langle \mathbf{a} \rangle}\, \bar{B} + \overline{\langle \mathbf{S}^{(0)} \rangle} \right), \tag{13.10}$$

where $\mathbf{I}$ denotes the identity matrix and $\mathbf{I}^{(0)}$ the initial Stokes vector. The *radiative transfer step* from $\mathbf{P}'$ to $\mathbf{P}$ is calculated, therefore $\mathbf{I}^{(0)}$ is the incoming radiation at $\mathbf{P}'$ into direction

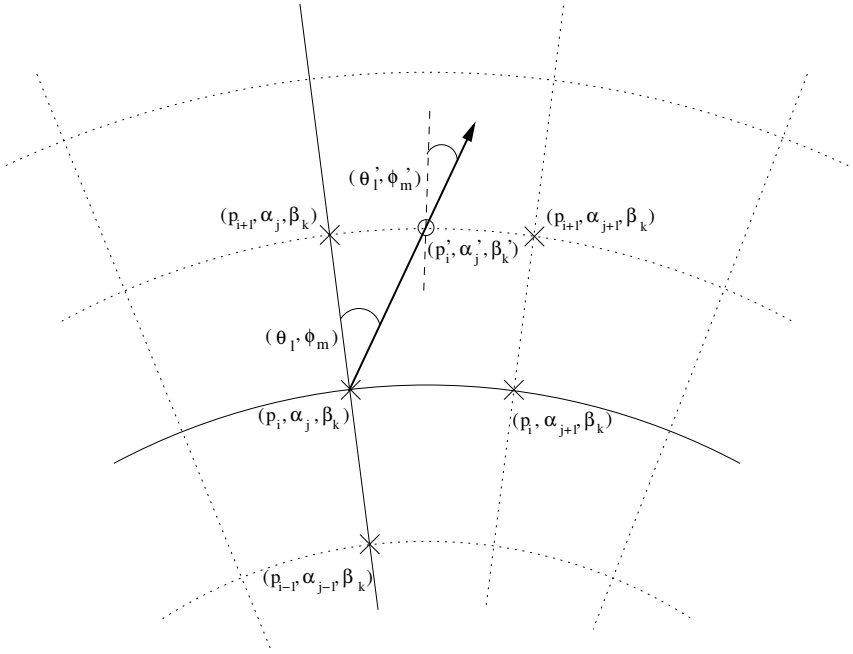

Figure 13.2: Path from a grid point ((P_i, α_j, β_k) - ($\times$)) to the intersection point ($(P_i', \alpha_j', \beta_k')$ - ($\circ$)) with the next grid cell boundary. Viewing direction is specified by (ψ_l, ω_m) at ($\times$) or (ψ_l', ω_m') at ($\circ$).

(ψ_l', ω_m'), which is the first guess field interpolated on $\mathbf{P}'$. This radiative transfer step calculation is done for all points inside the cloud box in all directions. The resulting set of Stokes vectors ($\mathbf{I}^{(1)}$ for all points in all directions) is the first order iteration field $\mathcal{I}^{(1)}$:

$$\mathcal{I}^{(1)} = \left\{ \mathbf{I}^{(1)}_{ijklm} \right\}. \tag{13.11}$$

The first order iteration field is stored in a variable reserved for the *new radiation field*.

In the *convergence test* the *new radiation field* is compared to the *old radiation field*. For the difference field, the absolute values of all Stokes vector elements for all cloud box positions are calculated. If one of the differences is larger than a requested accuracy limit, the convergence test is not fulfilled. The user can define different convergence limits for the different Stokes components.

If the convergence test is not fulfilled, the first order iteration field is copied to the variable holding the *old radiation field*, and is then used to evaluate again the scattering integral at all cloud box points:

$$\left\langle \mathbf{S}^{(1)}_{ijklm} \right\rangle = \int_{4\pi} \mathrm{d}\mathbf{n}' \left\langle \mathbf{Z} \right\rangle \mathbf{I}^{(1)}_{ijklm}. \tag{13.12}$$

The second order iteration field

$$\mathcal{I}^{(2)} = \left\{ \mathbf{I}^{(2)}_{ijklm} \right\}, \tag{13.13}$$

is obtained by solving

$$\frac{\mathrm{d}\mathbf{I}^{(2)}}{\mathrm{d}s} = -\overline{\langle \mathbf{K} \rangle} \mathbf{I}^{(2)} + \overline{\langle \mathbf{a} \rangle}\, \bar{B} + \overline{\langle \mathbf{S}^{(1)} \rangle}, \tag{13.14}$$

for all cloud box points in all directions. This equation contains already the averaged values and is valid for specified positions and directions.

As long as the convergence test is not fulfilled the scattering integral fields and higher order iteration fields are calculated alternately.

We can formulate a differential equation for the n-th order iteration field. The scattering integrals are given by

$$\left\langle \mathbf{S}_{ijklm}^{(n-1)} \right\rangle = \int_{4\pi} \mathrm{d}\mathbf{n}' \, \langle \mathbf{Z} \rangle \, \mathbf{I}_{ijklm}^{(n-1)}, \tag{13.15}$$

and the differential equation for a specified grid point into a specified direction is

$$\frac{\mathrm{d}\mathbf{I}^{(n)}}{\mathrm{d}s} = -\overline{\langle \mathbf{K} \rangle} \mathbf{I}^{(n)} + \overline{\langle \mathbf{a} \rangle} \, \bar{B} + \overline{\left\langle \mathbf{S}^{(n-1)} \right\rangle}. \tag{13.16}$$

Thus the *n-th order iteration field*

$$\mathcal{I}^{(n)} = \left\{ \mathbf{I}_{ijklm}^{(n)} \right\}, \tag{13.17}$$

is given by

$$\mathbf{I}^{(n)} = e^{-\overline{\langle \mathbf{K} \rangle} s} + \cdot \mathbf{I}^{(n-1)} (\mathbf{I} - e^{-\overline{\langle \mathbf{K} \rangle} s}) \overline{\langle \mathbf{K} \rangle}^{-1} (\overline{\langle \mathbf{a} \rangle} \, \bar{B} + \overline{\left\langle \mathbf{S}^{(n-1)} \right\rangle}), \tag{13.18}$$

for all cloud box points and all directions defined in the numerical grids.

If the convergence test

$$\left| \mathbf{I}_{ijklm}^{(N)} \left(P_i, \alpha_j, \beta_k, \psi_l, \omega_m \right) - \mathbf{I}_{ijklm}^{(N-1)} \left(P_i, \alpha_j, \beta_k, \psi_l, \omega_m \right) \right| < \epsilon, \tag{13.19}$$

is fulfilled, a solution to the vector radiative transfer equation (21.35) has been found:

$$\mathcal{I}^{(N)} = \left\{ \mathbf{I}_{ijklm}^{(N)} \right\}. \tag{13.20}$$

13.1.3 Scalar radiative transfer equation solution

In analogy to the *scattering integral* vector field the scalar scattering integral field is obtained:

$$\left\langle S_{ijklm}^{(0)} \right\rangle = \int_{4\pi} \mathrm{d}\mathbf{n}' \, \langle Z_{11} \rangle \, I_{ijklm}^{(0)}. \tag{13.21}$$

The *scalar radiative transfer* equation (21.45) with a fixed scattering integral is

$$\frac{\mathrm{d}I^{(1)}}{\mathrm{d}s} = - \langle K_{11} \rangle \, I^{(1)} + \langle a_1 \rangle \, B + \left\langle S^{(0)} \right\rangle. \tag{13.22}$$

Assuming constant coefficients this equation is solved analytically after averaging extinction coefficients, absorption coefficients, scattering vectors and the temperature. The averaging procedure is done analogously to the procedure described for solving the VRTE. The solution of the averaged differential equation is

$$I^{(1)} = I^{(0)} e^{-\overline{\langle K_{11} \rangle} s} + \frac{\overline{\langle a_1 \rangle} \, \bar{B} + \overline{\left\langle S^{(0)} \right\rangle}}{\overline{\langle K_{11} \rangle}} \left(1 - e^{-\overline{\langle K_{11} \rangle} s} \right), \tag{13.23}$$

where $I^{(0)}$ is obtained by interpolating the initial field, and $\overline{\langle K_{11} \rangle}$, $\overline{\langle a_1 \rangle}$, $\bar{B}$ and $\overline{\langle S^{(0)} \rangle}$ are the averaged values for the extinction coefficient, the absorption coefficient, the Planck function and the scattering integral respectively. Applying this equation leads to the first iteration scalar intensity field, consisting of the intensities $I^{(1)}$ at all points in the cloud box for all directions.

As the solution to the vector radiative transfer equation, the solution to the scalar radiative transfer equation is found numerically by the same iterative method. The convergence test for the scalar equation compares the values of the calculated intensities of two successive iteration fields.

13.1.4 Single scattering approximation

The DOIT method uses the single scattering approximation, which means that for one propagation path step the optical depth is assumed to be much less than one so that multiple-scattering can be neglected along this propagation path step. It is possible to choose a rather coarse grid inside the cloud box. The user can define a limit for the maximum propagation path step length. If a propagation path step from one grid cell to the intersection point with the next grid cell boundary is greater than this value, the path step is divided in several steps such that all steps are less than the maximum value. The user has to make sure that the optical depth due to cloud particles for one propagation path sub-step is is sufficiently small to assume single scattering. The maximum optical depth due to ice particles is

$$\tau_{max} = \langle \mathbf{K}^p \rangle \cdot \Delta s, \tag{13.24}$$

where Δs is the length of a propagation path step. In all simulations presented in *Emde* [2005], $\tau_{max} \ll 0.01$ is assumed. This threshold value is also used in *Czekala* [1999]. The radiative transfer calculation is done along the propagation path through one grid cell. All coefficients of the VRTE are interpolated linearly on the propagation path points.

13.2 Sequential update

In the previous Section 13.1 the iterative solution method for the VRTE has been described. For each grid point inside the cloud box the intersection point with the next grid cell boundary is determined in each viewing direction. After that, all the quantities involved in the VRTE are interpolated onto this intersection point. As described in the sections above, the intensity field of the previous iteration is taken to obtain the Stokes vector at the intersection point. Suppose that there are N pressure levels inside the cloud box. If the radiation field is updated taking into account for each grid point only the adjacent grid cells, at least N-1 iterations are required until the scattering effect from the lower-most pressure level has propagated throughout the cloud box up to the uppermost pressure level. From these considerations, it follows, that the number of iterations depends on the number of grid points inside the cloud box. This means that the original method is very ineffective where a fine resolution inside the cloud box is required to resolve the cloud inhomogeneities.

A solution to this problem is the “sequential update of the radiation field”, which is shown schematically in Figure 13.3. For simplicity it will be explained in detail for a 1D cloud box. We divide the update of the radiation field, i.e., the radiative transfer step calculations for all positions and directions inside the cloud box, into three parts: Update for

"up-looking" zenith angles ($0° \leq \psi_{\rm up} \leq 90°$), for "down-looking" angles ($\psi_{\rm limit} \leq \psi_{\rm down} \leq 180°$) and for "limb-looking" angles ($90° < \psi_{\rm limb} < \psi_{\rm limit}$). The "limb-looking" case is needed, because for angles between $90°$ and $\psi_{\rm limit}$ the intersection point is at the same pressure level as the observation point. The limiting angle $\psi_{\rm limit}$ is calculated geometrically. Note that the propagation direction of the radiation is opposite to the viewing direction or the direction of the line of sight, which is indicated by the arrows. In the 1D case the radiation field is a set of Stokes vectors each of which depend upon the position and direction:

$$\mathcal{I} = \{\mathbf{I}\left(P_i, \psi_l\right)\}. \tag{13.25}$$

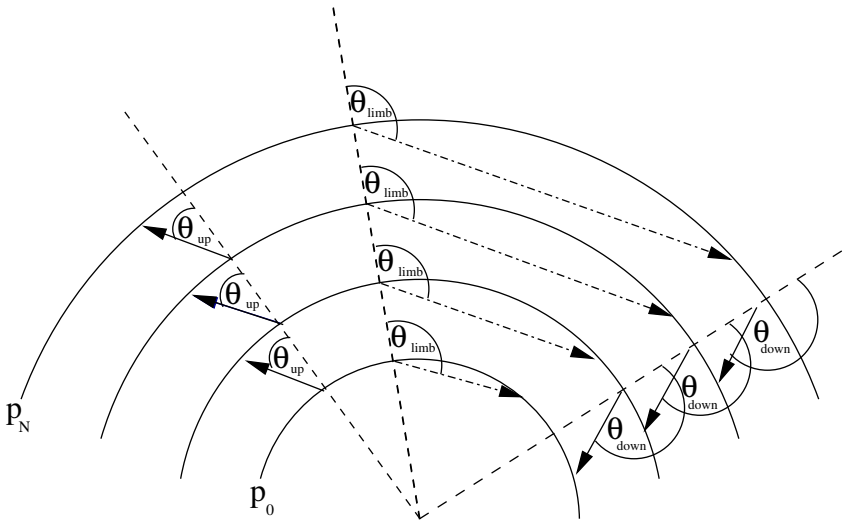

Figure 13.3: Schematic of the sequential update (1D) showing the three different parts: "up-looking" corresponds to zenith angles $\psi_{\rm up}$, "limb-looking" corresponds to $\psi_{\rm limb}$ "down-looking" corresponds to $\psi_{\rm down}$.

The *boundary condition* for the calculation is the incoming radiation field on the cloud box boundary $\mathcal{I}^{bd}$:

$$\begin{aligned}\mathcal{I}^{bd} = \{\mathbf{I}\left(P_i, \psi_l\right)\} \text{ where} \qquad & P_i = P_N \,\forall\, \psi_l \in [0, \psi_{\rm limit}] \\ & P_i = P_0 \,\forall\, \psi_l \in (\psi_{\rm limit}, 180°],\end{aligned} \tag{13.26}$$

where P_0 and P_N are the pressure coordinates of the lower and upper cloud box boundaries respectively. For down-looking directions, the intensity field at the lower-most cloud box boundary and for up- and limb-looking directions the intensity field at the uppermost cloud box boundary are the required boundary conditions respectively.

13.2.1 Up-looking directions

The first step of the sequential update is to calculate the intensity field for the pressure coordinate P_{N-1}, the pressure level below the uppermost boundary, for all up-looking directions. Radiative transfer steps are calculated for paths starting at the uppermost boundary and propagating to the $(N-1)$ pressure level. The required input for this radiative transfer

step are the averaged coefficients of the uppermost cloud box layer and the Stokes vectors at the uppermost boundary for all up-looking directions. These are obtained by interpolating the boundary condition $\mathcal{I}^{bd}$ on the appropriate zenith angles. Note that the zenith angle of the propagation path for the observing direction ψ_l does not equal ψ_l' at the intersection point due to the spherical geometry. If ψ_l is close to 90° this difference is most significant.

To calculate the intensity field for the pressure coordinate P_{N-2}, we repeat the calculation above. We have to calculate a radiative transfer step from the $(N-1)$ to the $(N-2)$ pressure level. As input we need the interpolated intensity field at the $(N-1)$ pressure level, which has been calculated in the last step.

For each pressure level $(m-1)$ we take the interpolated field of the layer above $(\mathcal{I}(P_m)^{(1)})$. Using this method, the scattering influence from particles in the upper-most cloud box layer can propagate during one iteration down to the lower-most layer. This means that the number of iterations does not scale with the number of pressure levels, which would be the case without sequential update.

The radiation field at a specific point in the cloud box is obtained by solving Equation (13.10). For up-looking directions at position P_{m-1} we may write:

$$\begin{aligned}\mathbf{I}\left(P_{m-1},\psi_{\mathrm{up}}\right)^{(1)} &= e^{-\overline{\langle \mathbf{K}(\psi_{\mathrm{up}})\rangle} s}\mathbf{I}\left(P_m,\psi_{\mathrm{up}}\right)^{(1)} \\ + \left(\mathbf{I}-e^{-\overline{\langle \mathbf{K}(\psi_{\mathrm{up}})\rangle} s}\right)\overline{\langle \mathbf{K}(\psi_{\mathrm{up}})\rangle}^{-1} &\left(\overline{\langle \mathbf{a}(\psi_{\mathrm{up}})\rangle}\,\bar{B}+\overline{\left\langle \mathbf{S}\left(\psi_{\mathrm{up}}\right)^{(0)}\right\rangle}\right). \end{aligned} \tag{13.27}$$

For simplification we write

$$\mathbf{I}(P_{m-1},\psi_{\mathrm{up}})^{(1)} = \mathbf{A}(\psi_{\mathrm{up}})\mathbf{I}\left(P_m,\psi_{\mathrm{up}}\right)^{(1)} + \mathbf{B}(\psi_{\mathrm{up}}). \tag{13.28}$$

Solving this equation sequentially, starting at the top of the cloud and finishing at the bottom, we get the updated radiation field for all up-looking angles.

$$\mathcal{I}(P_i,\psi_{\mathrm{up}})^{(1)} = \left\{\mathbf{I}^{(1)}\left(P_i,\psi_l\right)\right\} \qquad \forall\ \psi_l \in [0,90^\circ]. \tag{13.29}$$

13.2.2 Down-looking directions

The same procedure is done for down-looking directions. The only difference is that the starting point is the lower-most pressure level P_1 and the incoming clear sky field at the lower cloud box boundary, which is interpolated on the required zenith angles, is taken as boundary condition. The following equation is solved sequentially, starting at the bottom of the cloud box and finishing at the top:

$$\mathbf{I}(P_m,\psi_{\mathrm{down}})^{(1)} = \mathbf{A}(\psi_{\mathrm{down}})\mathbf{I}\left(P_{m-1},\psi_{\mathrm{down}}\right)^{(1)} + \mathbf{B}(\psi_{\mathrm{down}}). \tag{13.30}$$

This yields the updated radiation field for all down-looking angles.

$$\mathcal{I}(P_i,\psi_{\mathrm{down}})^{(1)} = \left\{\mathbf{I}^{(1)}\left(P_i,\psi_l\right)\right\} \qquad \forall\ \psi_l \in [\psi_{\mathrm{limit}},180^\circ]. \tag{13.31}$$

13.2.3 Limb directions

A special case for limb directions, which correspond to angles slightly above 90° had to be implemented. If the tangent point is part of the propagation path step, the intersection point is exactly at the same pressure level as the starting point. In this case the linearly

interpolated clear sky field is taken as input for the radiative transfer calculation, because we do not have an already updated field for this pressure level:

$$\mathbf{I}(P_m, \psi_{\text{limb}})^{(1)} = \mathbf{A}(\psi_{\text{limb}})\mathbf{I}\,(P_m, \psi_{\text{limb}})^{(0)} + \mathbf{B}(\psi_{\text{limb}}) \tag{13.32}$$

By solving this equation the missing part of the updated radiation field is obtained

$$\mathcal{I}(P_i, \psi_{\text{limb}})^{(1)} = \{\mathbf{I}\,(P_i, \psi_l)\} \qquad \forall\ \psi_l \in]90^\circ, \psi_{\text{limit}}[\tag{13.33}$$

For all iterations the sequential update is applied. Using this method the number of iterations depends only on the optical thickness of the cloud or on the number of multiple-scattering events, not on the number of pressure levels.

13.3 Numerical Issues

13.3.1 Grid optimization and interpolation

The accuracy of the DOIT method depends very much on the discretization of the zenith angle. The reason is that the intensity field strongly increases at about $\psi = 90^\circ$. For angles below 90° ("up-looking" directions) the intensity is very small compared to angles above 90° ("down-looking" directions), because the thermal emission from the lower atmosphere and from the ground is much larger than thermal emission from trace gases in the upper atmosphere. Figure 13.4 shows an example intensity field as a function of zenith angle for different pressure levels inside a cloud box, which is placed from 7.3 to 12.7 km altitude, corresponding to pressure limits of 411 hPa and 188 hPa respectively. The cloud box includes 27 pressure levels. The frequency of the sample calculation was 318 GHz. A midlatitude-summer scenario including water vapor, ozone, nitrogen and oxygen was used. The atmospheric data was taken from the FASCOD [*Anderson et al.*, 1986] and the spectroscopic data was obtained from the HITRAN database [*Rothman et al.*, 1998]. For simplicity this 1D set-up was chosen for all sample calculations in this section. As the intensity (or the Stokes vector) at the intersection point of a propagation path is obtained by interpolation, large interpolation errors can occur for zenith angles of about 90° if the zenith angle grid discretization is too coarse. Taking a very fine equidistant zenith angle grid leads to very long computation times. Therefore a zenith angle grid optimization method is required.

For the computation of the scattering integral it is possible to take a much coarser zenith angle resolution without losing accuracy. It does not make sense to use the zenith angle grid, which is optimized to represent the radiation field with a certain accuracy. The integrand is the product of the phase matrix and the radiation field. The peaks of the phase matrices can be at any zenith angle, depending on the incoming and the scattered directions. The multiplication smooths out both the radiation field increase at 90° and the peaks of the phase matrices. Test calculations have shown that an increment of 10° is sufficient. Taking the equidistant grid saves the computation time of the scattering integral to a very large extent, because much less grid points are required.

Zenith angle grid optimization

As a reference field for the grid optimization the DOIT method is applied for an empty cloud box using a very fine zenith angle grid. The grid optimization routine finds a reduced

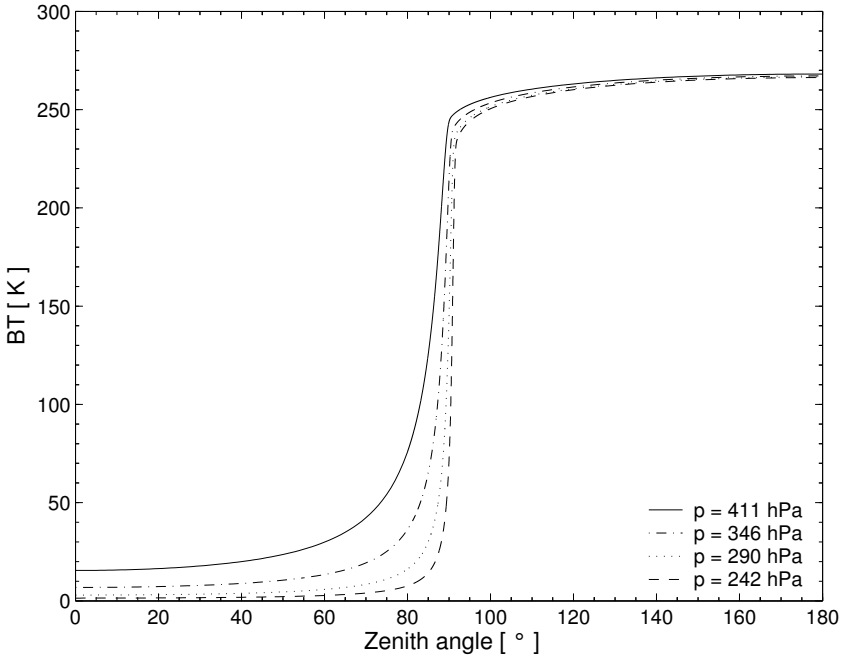

Figure 13.4: Intensity field for different pressure levels.

zenith angle grid which can represent the intensity field with the desired accuracy. It first takes the radiation at 0° and 180° and interpolates between these two points on all grid points contained in the fine zenith angle grid for all pressure levels. Then the differences between the reference radiation field and the interpolated field are calculated. The zenith angle grid point, where the difference is maximal is added to 0° and 180°. After that the radiation field is interpolated between these three points forming part of the reduced grid and again the grid point with the maximum difference is added. Using this method more and more grid points are added to the reduced grid until the maximum difference is below a requested accuracy limit.

The top panel of Figure 13.5 shows the clear sky radiation in all viewing directions for a sensor located at 13 km altitude. This result was obtained with a switched-off cloud box. The difference between the clear sky part of the ARTS model and the scattering part is that in the clear sky part the radiative transfer calculations are done along the line of sight of the instrument whereas inside the cloud box the RT calculations are done as described in the previous section to obtain the full radiation field inside the cloud box. In the clear sky part the radiation field is not interpolated, therefore we can take the clear sky solution as the exact solution.

The interpolation error is the relative difference between the exact clear sky calculation (cloud box switched off) and the clear sky calculation with empty cloud box. The bottom panels of Figure 13.5 show the interpolation errors for zenith angle grids optimized with three different accuracy limits (0.1%, 0.2% and 0.5%.). The left plot shows the critical region close to 90°. For a grid optimization accuracy of 0.5% the interpolation error becomes very large, the maximum error is about 8%. For grid accuracies of 0.2% and 0.1% the maximum interpolation errors are about 0.4% and 0.2% respectively. However for most angles

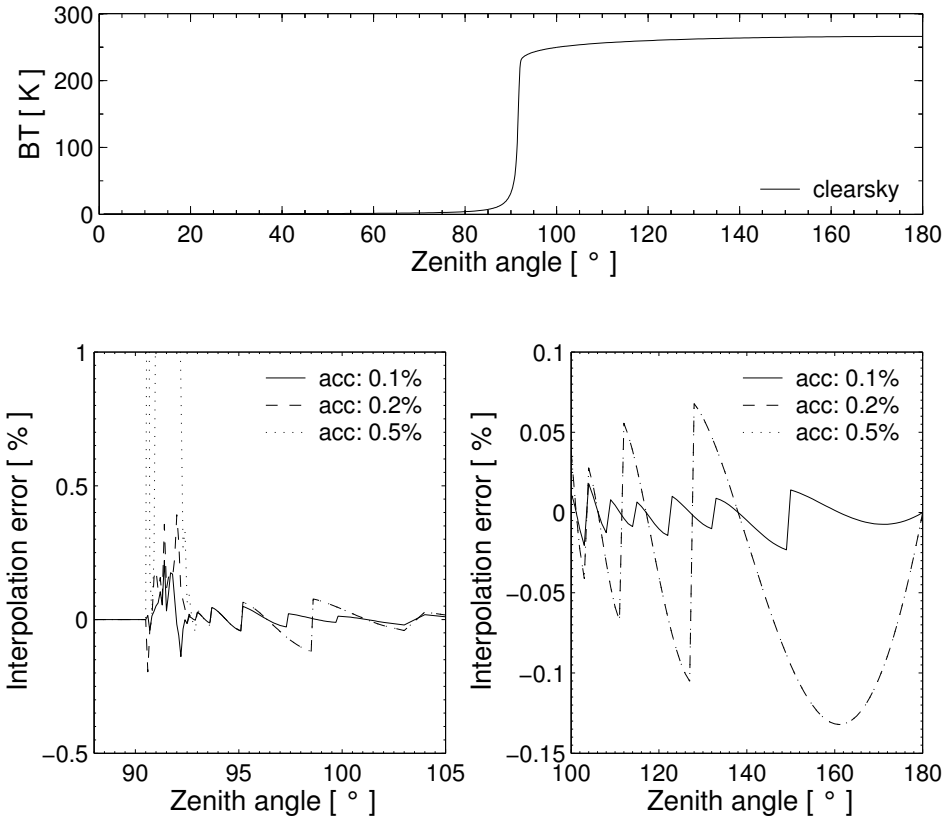

Figure 13.5: Interpolation errors for different grid accuracies. Top panel: Clear sky radiation simulated for a sensor at an altitude of 13 km for all viewing directions. Bottom left: Grid optimization accuracy for limb directions. Bottom right: Grid optimization accuracy for down-looking directions.

it is below 0.2%, for all three cases. For down-looking directions from 100° to 180° the interpolation error is at most 0.14% for grid accuracies of 0.2% and 0.5% and for a grid accuracy of 0.1% it is below 0.02%.

Interpolation methods

Two different interpolation methods can be chosen in ARTS for the interpolation of the radiation field in the zenith angle dimension: linear interpolation or three-point polynomial interpolation. The polynomial interpolation method produces more accurate results provided that the zenith angle grid is optimized appropriately. The linear interpolation method on the other hand is safer. If the zenith angle grid is not optimized for polynomial interpolation one should use the simpler linear interpolation method. Apart from the interpolation of the radiation field in the zenith angle dimension linear interpolation is used everywhere in the model. Figure 13.6 shows the interpolation errors for the different interpolation methods. Both calculations are performed on optimized zenith angle grids, for polynomial interpolation 65 grid points were required to achieve an accuracy of 0.1% and for linear interpolation 101 points were necessary to achieve the same accuracy. In the region of about 90° the

interpolation errors are below 1.2% for linear interpolation and below 0.2% for polynomial interpolation. For the other down-looking directions the differences are below 0.08% for linear and below 0.02% for polynomial interpolation. It is obvious that polynomial interpolation gives more accurate results. Another advantage is that the calculation is faster because less grid points are required, although the polynomial interpolation method itself is slower than the linear interpolation method. Nevertheless, we have implemented the polynomial interpolation method so far only in the 1D model. In the 3D model, the grid optimization needs to be done over the whole cloud box, where it is not obvious that one can save grid points. Applying the polynomial interpolation method using non-optimized grids can yield much larger interpolation errors than the linear interpolation method.

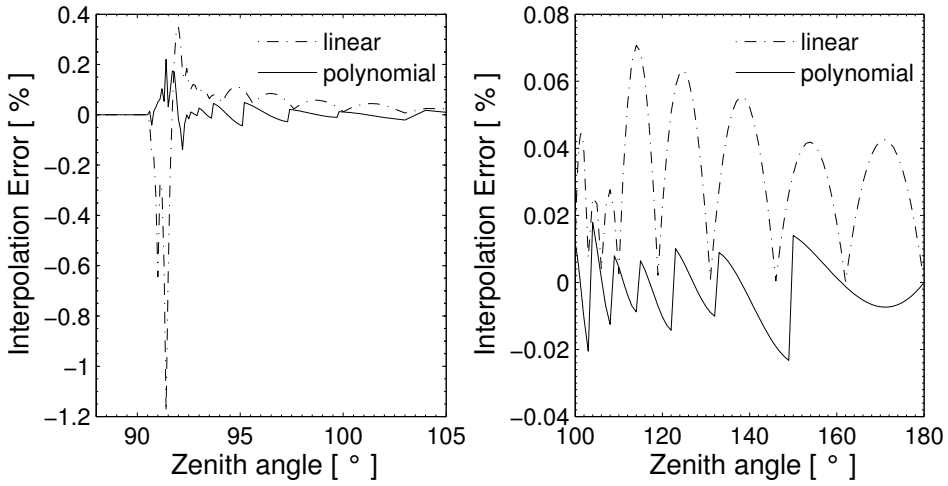

Figure 13.6: Interpolation errors for polynomial and linear interpolation.

Error estimates

The interpolation error for scattering calculations can be estimated by comparison of a scattering calculation performed on a very fine zenith angle grid (resolution 0.001° from 80° to 100°) with a scattering calculation performed on an optimized zenith angle grid with 0.1% accuracy. The cloud box used in previous test calculations is filled with spheroidal particles with an aspect ratio of 0.5 from 10 to 12 km altitude. The ice mass content is assumed to be $4.3 \cdot 10^{-3}$ g/m^3 at all pressure levels. An equal volume sphere radius of 75 μm is assumed. The particles are either completely randomly oriented (p20) or azimuthally randomly oriented (p30) (cf. Section 12.2.3). The top panels of Figure 13.7 show the interpolation errors of the intensity. For both particle orientations the interpolation error is in the same range as the error for the clear sky calculation, below 0.2 K. The bottom panels show the interpolation errors for Q. For the randomly oriented particles the error is below 0.5%. For the horizontally aligned particles with random azimuthal orientation it goes up to 2.5% for a zenith angle of about 91.5°. It is obvious that the interpolation error for Q must be larger than that for I because the grid optimization is accomplished using only the clear-sky field, where the polarization is zero. Only the limb directions about 90° are problematic, for other down-looking directions the interpolation error is below 0.2%.

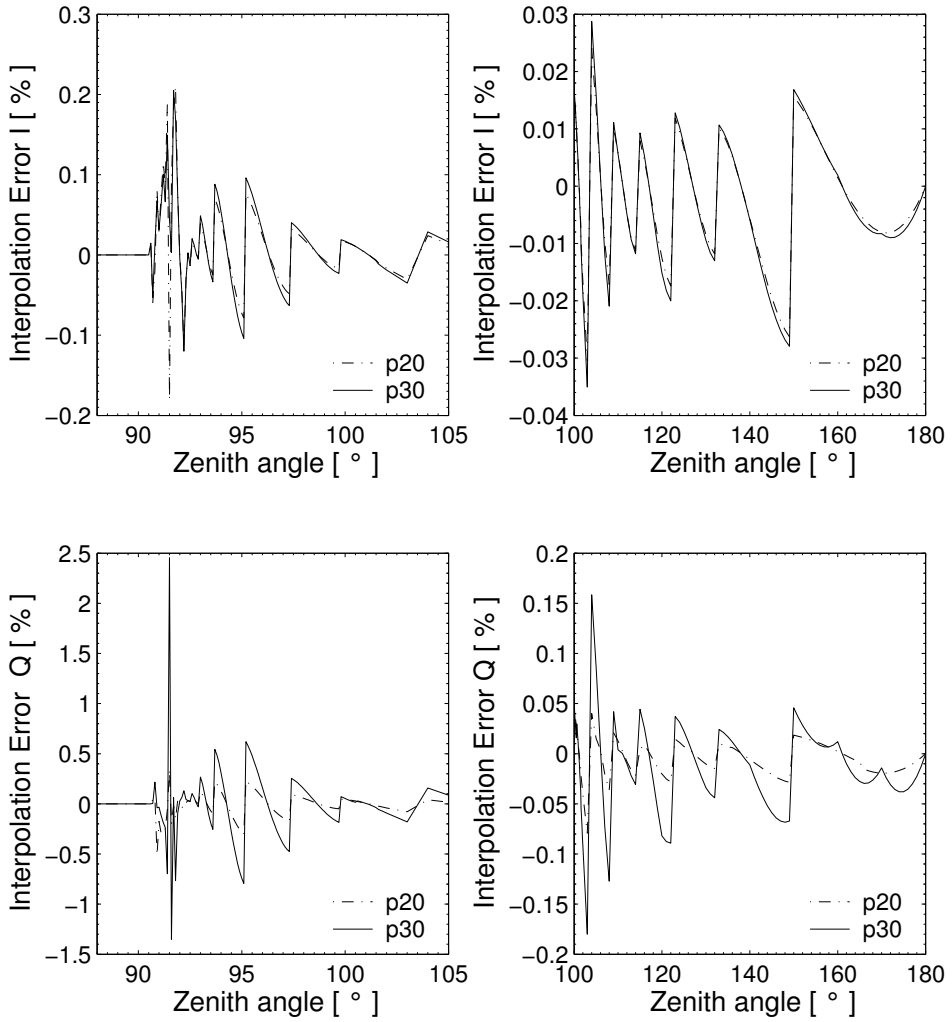

Figure 13.7: Interpolation errors for a scattering calculation. Left panels: Interpolation errors for limb directions. Right panels: Interpolation errors for down-looking directions. Top: Intensity I, Bottom: Polarization difference Q

13.4 Implementation

The workspace methods required for DOIT calculations are implemented in the files `m_scatrte.cc`, `m_cloudbox.cc` and `m_optproperties.cc`.

13.4.1 1D control file example

This example demonstrates how to set up a 1D DOIT calculation. A full running controlfile example for a DOIT calculation can be found in the ARTS package in the tests/DOIT directory. The file is called `TestDOIT.arts`. For detailed descriptions of the workspace methods and variables please refer also to the online help (`arts -d ...`).

13.4.2 DOIT frame

As a first step for a DOIT calculation we have to calculate the incoming field on the boundary of the cloudbox. This is done using the workspace method `CloudboxGetIncoming`:

```
CloudboxGetIncoming
```

The next step is the initialization of variables required for a DOIT calculation using `DoitInit`:

```
DoitInit
```

The grid discretization plays a very significant role in discrete ordinate methods. In spherical geometry the zenith angular grid is of particular importance (cf. Section 13.3.1). The angular discretization is defined in the workspace method `DoitAngularGridsSet`:

```
DoitAngularGridsSet( doit_za_grid_size,
                     scat_aa_grid, scat_za_grid,
                     19, 10, "doit_za_grid_opt.xml" )
```

For down-looking geometries it is sufficient to define the generic inputs:
`N_za_grid` Number of grid points in zenith angle grid, recommended value: 19
`N_aa_grid` Number of grid points in azimuth angle grid, recommended value: 37
From these numbers equally spaced grids are created and stored in the work space variables `scat_za_grid` and `scat_aa_grid`.

For limb simulations it is important to use an optimized zenith angle grid with a very fine resolution about 90° for the RT calculations. Such a grid can be generated using the workspace method `doit_za_grid_optCalc`. Please refer to the online documentation of this method. The filename of the optimized zenith angle grid can be given as a generic input. If a filename is given, the equidistant grid is taken for the calculation of the scattering integrals and the optimized grid is taken for the radiative transfer part. Otherwise, if no filename is specified (`za_grid_opt_file = ""`) the equidistant grid is taken for the calculation of the scattering integrals and for the radiative transfer calculations. This option makes sense for down-looking cases to speed up the calculation.

The main agenda for a DOIT calculation is `doit_mono_agenda`. The agenda is executed by the workspace method `ScatteringDoit`:

```
ScatteringDoit
```

The DOIT main agenda

The agenda doit_main_agenda requires the incoming clearsky field on the cloudbox boundary as an input and gives as output the scattered field on the cloudbox boundary if the sensor is placed outside the cloudbox or the full scattered field in the cloudbox if the sensor is placed inside the cloudbox.

```
AgendaSet( doit_mono_agenda ){
 # Prepare scattering data for DOIT calculation (Optimized method):
   DoitScatteringDataPrepare
 # Alternative method (needs less memory):
 # scat_data_monoCalc
 # Set first guess field:
   doit_i_fieldSetClearsky
 # Perform iterations: 1. scattering integral. 2. RT calculations with
 #   fixed scattering integral field, 3. convergence test
   doit_i_fieldIterate
 # Put solution into interface for clearsky calculation
   DoitCloudboxFieldPut
 }
```

The first method DoitScatteringDataPrepare reads the single scattering data and interpolates it on the requested frequency. It also performs the transformation from the scattering frame into the laboratory frame. Alternatively the method scat_data_monoCalc can be used. In this case only the frequency interpolation is done and the transformations are done later. The advantage is that this method needs less memory. For 1D calculation it is recommended to use DoitScatteringDataPrepare because it is much more efficient.

The method doit_i_fieldSetClearsky interpolates the incoming radiation field on all points inside the cloudbox to obtain the initial field (doit_i_field) for the DOIT calculation. As a test one can alternatively start with a constant radiation field using the method doit_i_fieldSetConst.

The iteration is performed in the method doit_i_fieldIterate, which includes the calculation of the scattering integral field (doit_scat_field), the radiative transfer calculations in the cloudbox with fixed scattering integral and the convergence test.

After convergence is obtained the radiation field inside the cloudbox is stored in the interface variable scat_i_p if the sensor is located outside the cloudbox, or in the variable doit_i_field1D_spectrum if the sensor is located inside the cloudbox. This is done by the workspace method DoitCloudboxFieldPut. In contrast to doit_i_field the interface variables include an additional dimension for the frequency.

Agendas used in doit_i_fieldIterate

There are several methods which can be used in doit_i_fieldIterate, for instance for the calculation of the scattering integral. The methods are selected in the control-file by defining several agendas.

Calculation of the scattering integral: To calculate the scattering integral (Equation (13.7)) the phase matrix (pha_mat) is required. How the phase matrix is calculated is defined in the agenda pha_mat_spt_agenda:

```
# Calculation of the phase matrix
AgendaSet( pha_mat_spt_agenda ){
 # Optimized option:
   pha_mat_sptFromDataDOITOpt
 # Alternative option:
 #  pha_mat_sptFromMonoData
}
```

If in doit_mono_agenda the optimized method DoitScatteringDataPrepare is used we have to use here the corresponding method pha_mat_sptFromDataDOITOpt. Otherwise we have to use pha_mat_sptFromMonoData.

To do the integration itself, we have to define doit_scat_field_agenda:

```
AgendaSet( doit_scat_field_agenda ){
 doit_scat_fieldCalcLimb
 # Alternative:
 # doit_scat_fieldCalc
}
```

Here we have two options. One is doit_scat_fieldCalcLimb, which should be used for limb simulations, for which we need a fine zenith angle grid resolution to represent the radiation field. This method has to be used if a zenith angle grid file is given in DoitAngularGridsSet. The scattering integral can be calculated on a coarser grid resolution, hence in doit_scat_fieldCalcLimb, the radiation field is interpolated on the equidistant angular grids specified in DoitAngularGridsSet by the generic inputs Nza and Naa. Alternatively, one can use doit_scat_fieldCalc, where this interpolation is not performed. This function is efficient for simulations in up- or down-looking geometry, where the we do not need the fine zenith angle grid resolution about 90°.

Radiative transfer with fixed scattering integral term: With a fixed scattering integral field the radiative transfer equation can be solved (Equation (13.9)). The workspace method to be used for this calculation is defined in doit_rte_agenda. The most efficient and recommended workspace method is doit_i_fieldUpdateSeq1D where the sequential update which is described in Section 13.2 is applied. The workspace method doit_i_fieldUpdate1D does the same calculation without sequential update and is therefore much less efficient because the number of iterations depends in this case on the number of pressure levels in the cloudbox. Other options are to use a plane-parallel approximation implemented in the workspace method doit_i_fieldUpdateSeq1DPP. This method is not much more efficient than doit_i_fieldUpdateSeq1D, therefore it is usually better to use doit_i_fieldUpdateSeq1D since it is more accurate.

```
AgendaSet( doit_rte_agenda ){
     doit_i_fieldUpdateSeq1D
   # Alternatives:
```

```
   # doit_i_fieldUpdateSeq1DPP
   # i_fieldUpdate1D
   }
```

The optical properties of the particles, i.e., extinction matrix and absorption vector are required for solving the radiative transfer equation. How they are calculated is specified in `spt_calc_agenda`. The workspace method `opt_prop_sptFromMonoData` requires that the raw data is already interpolated on the frequency of the monochromatic calculation. This requirement is fulfilled when `DoitScatteringDataPrepare` of `scat_data_monoCalc` is executed before `doit_i_fieldIterate` (see Section 13.4.2). The work space method `ext_matAddPart` and `abs_vecAddPart` are used to extract the absorption vector `abs_vec` and extinction matrix `ext_mat` from the workspace variable `opt_prop_spt`. The gas absorption is added internally.

```
AgendaSet( spt_calc_agenda ){
   opt_prop_sptFromMonoData
}
AgendaSet( opt_prop_part_agenda ){
   ext_matInit
   abs_vecInit
   ext_matAddPart
   abs_vecAddPart
}
```

Convergence test: After the radiative transfer calculations with a fixed scattering integral field are complete the newly obtained radiation field is compared to the old radiation field by a convergence test. The functions and parameters for the convergence test are defined in the agenda `doit_conv_test_agenda`. There are several options. The workspace methods `doit_conv_flagAbsBT` and `doit_conv_flagAbs` compare the absolute differences of the radiation field element-wise as described in Equation (13.19). The convergence limits are specified by the generic input `epsilon` which specifies the convergence limit. A limit must be given for each Stokes component. In `doit_conv_flagAbsBT` the limits must be specified in Rayleigh Jeans brightness temperatures whereas in `doit_conv_flagAbs` they must be defined in the basic radiance unit ([W/(m^{2}Hz sr)]. Another option is to perform a least square convergence test using the workspace method `doit_conv_flagLsq`. Test calculations have shown that this test is not safe, therefore the least square convergence test should only be used for test purposes.

```
AgendaSet( doit_conv_test_agenda) {
    doit_conv_flagAbsBT( doit_conv_flag, doit_iteration_counter,
                         doit_i_field, doit_i_field_old,
                         f_grid, f_index,
                         [0.1, 0.01, 0.01, 0.01] )
    # Alternative: Give limits in radiances
    #   doit_conv_flagAbs( doit_conv_flag, doit_iteration_counter,
    #                      doit_i_field, doit_i_field_old,
    #                      [0.1e-15, 0.1e-18, 0.1e-18, 0.1e-18] )
```

```
    #
    # If you want to look at several iteration fields, for example
    # to investigate the convergence behavior, you can use
    # the following workspace method:
    # DoitWriteIterationFields( doit_iteration_counter, doit_i_field,
    #                           [2, 4] )
}
```

13.4.3 Propagation of the DOIT result towards the sensor

In order to propagate the result of the scattering calculation towards the sensor, the fields needs to be interpolated on the direction of the sensor's line of sight. This is done in the workspace method `iyInterpCloudboxField`, which has to be put into the agenda `iy_cloudbox_agenda`:

```
AgendaSet( iy_cloudbox_agenda ){
    iyInterpCloudboxField
}
```

13.4.4 3D DOIT calculations

The DOIT method is implemented for 1D and 3D spherical atmospheres, but it is strongly recommended to use it only for 1D calculations, because there are several numerical difficulties related to the grid discretizations. It is difficult to find appropriate discretizations to get sufficiently accurate results in reasonable computation time. Therefore only experienced ARTS users should use DOIT for 3D calculations only for smaller cloud scenarios. Please refer to the online documentation for the workspace method for 3D scattering calculations (`doit_i_fieldUpdateSeq3D`). All other workspace methods adapt automatically to the atmospheric dimensionality.

Chapter 14

Reversed Monte Carlo Scattering Module : ARTS-MC

14.1 Introduction

Much of the following has been taken from an article submitted to IEEE TGARS [Davis et al., 2004], although here there is a more detailed description of how the algorithm is implemented in ARTS, and how to use it.

Discrete ordinates (DOM) type methods are attractive when simulating the whole radiation field, but in the limb sounding - and other remote sensing cases, only a very limited subset of outward propagation paths are required. Also, for limb sounding simulations there is a strong variation in incoming radiance with zenith angles close to 90°. In DOM type models, this necessitates a very fine angular grid, which can be expensive.

A reversed Monte Carlo method was chosen for this study. A strong consideration here was that the simplicity of the Monte Carlo radiative transfer concept should translate to reduced development time. Also, reversed Monte Carlo methods allow all computational effort to be concentrated on calculating radiances for the desired line of sight, and the nature of Monte Carlo algorithms makes parallel computing trivial.

Among the available Backward Monte Carlo RT models, several do not allow a thermal source, or do not consider polarization fully (i.e. allowing a non-diagonal extinction matrix) (e.g. *Liu et al.* [1996]), and some consider neither (e.g. *Oikarinen et al.* [1999], *Ishimoto and Masuda* [2002]).

A useful reference for model development in this study, is the Backward-Forward Monte Carlo (BFMC) model described by *Liu et al.* [1996]. In BMFC photon paths are traced backwards from the sensor, with scattering angles and path lengths randomly chosen from probability density functions (PDF) determined by the scattering phase function, and a scalar extinction coefficient respectively. The phase matrices for every scattering event and scalar extinction are then sequentially applied to the source Stokes vector to give the Stokes vector contribution for each photon. As presented in *Liu et al.* [1996], the model is only applicable to cases where the extinction matrix is diagonal - that is, where there is macroscopically

---

History

300504 Created and written by Cory Davis.

isotropic and mirror-symmetric scattering media. This prompted Roberti and Kummerow *Roberti and Kummerow* [1999] to abandon the Backward-Forward Monte Carlo method and choose a modified Forward Monte Carlo model. However, in this study the attractive features of Liu's Backward-Forward model are retained by utilizing importance sampling, a well known technique in Monte Carlo integration. Importance sampling allows independent variables, in this case scattering angles and path lengths, to be sampled from any distribution as long as each contribution to the final integral is properly weighted.

14.2 Model

The radiative transfer model solves the vector radiative transfer equation (VRTE):

$$\frac{d\mathbf{I}(\mathbf{n})}{ds} = -\mathbf{K}(\mathbf{n})\mathbf{I}(\mathbf{n}) + \mathbf{K_a}(\mathbf{n})I_b(T) + \int_{4\pi} \mathbf{Z}(\mathbf{n},\mathbf{n}')\mathbf{I}(\mathbf{n}')d\mathbf{n}' \tag{14.1}$$

where $\mathbf{I}$ is the 4 element column vector of radiances $\mathbf{I} = [I, Q, U, V]^T$ with units ($\mathrm{Wm}^{-2}\mu\mathrm{m}^{-1}\mathrm{sr}^{-1}$). This will be referred to as the Stokes vector, although normally the Stokes vector is expressed in units of intensity. s is distance along direction $\mathbf{n}$ and I_b is the Planck radiance. $\mathbf{K}(\mathbf{n})$, $\mathbf{K_a}(\mathbf{n})$, and $\mathbf{Z}(\mathbf{n},\mathbf{n}')$ are the bulk extinction matrix, absorption coefficient vector and phase matrix of the medium respectively. For brevity these have been expressed as bulk optical properties, where individual single scattering properties have been multiplied by particle number density and averaged over all orientations and particle types. The argument $\mathbf{n}$ has been retained to signify that in general these properties depend on the direction of propagation.

To apply Monte Carlo integration to the problem, the VRTE needs to be expressed in integral form. (e.g. *Hochstadt* [1964])

$$\mathbf{I}(\mathbf{n},\mathbf{s_0}) = \mathbf{O}(\mathbf{u_0},\mathbf{s_0})\mathbf{I}(\mathbf{n},\mathbf{u_0}) + \int_{u_0}^{s_0} \mathbf{O}(\mathbf{s}',\mathbf{s_0}) \left(\mathbf{K_a}(\mathbf{n})I_b(T) + \int_{4\pi} \mathbf{Z}(\mathbf{n},\mathbf{n}')\mathbf{I}(\mathbf{n}')d\mathbf{n}'\right) ds' \tag{14.2}$$

, where $\mathbf{O}(\mathbf{s}',\mathbf{s})$ is the evolution operator defined by *Degl'Innocenti and Degl'Innocenti* [1985]. $\mathbf{u_0}$ is the point where the line of sight intersects the far boundary of the scattering domain, and $\mathbf{s_0}$ is the exit point where the outgoing Stokes vector is calculated. In general there is no closed form expression for $\mathbf{O}(\mathbf{s}',\mathbf{s})$. However, in cases where the extinction matrix is constant along a propagation path

$$\mathbf{O}(\mathbf{s}',\mathbf{s}) = \exp\left(-\mathbf{K}\Delta s\right) \tag{14.3}$$

In ARTS a propagation path consists of a set of coordinates indicating where the path intersects with grid surfaces. If the extinction matrix in the path segment between two such points is considered constant, $\mathbf{K} = (\mathbf{K_j} + \mathbf{K_{j+1}})/2$, the evolution operator between two arbitrary points $\mathbf{s_0}$ and $\mathbf{s}_N$ is

$$\mathbf{O}(\mathbf{s}_0,\mathbf{s}_N) = \mathbf{O}(\mathbf{s}_{N-1},\mathbf{s}_N)\mathbf{O}(\mathbf{s}_{N-2},\mathbf{s}_{N-1})\ldots \mathbf{O}(\mathbf{s}_1,\mathbf{s}_2)\mathbf{O}(\mathbf{s}_0,\mathbf{s}_1), \tag{14.4}$$

, where $\mathbf{O}(\mathbf{s_i}, \mathbf{s_{i+1}})$ is given by Eq. 14.3.

The numerical task is then to perform Monte Carlo integration on the integral on the right hand side of Eq. 14.2. The aim in importance sampling is to choose probability density functions (PDFs) for the independent variables that are as close as possible to being proportional to the integrand *Liu* [2001]. This concentrates computational effort on regions where the integrand is most significant and also reduces the variance in the contributions of each photon, thus reducing the number of photons and hence CPU time required to give a prescribed accuracy. Eq. 14.2 suggests that the PDF for sampling path length, where path length is the distance traced backwards from the sensor, $\Delta s = |\mathbf{s} - \mathbf{s}'|$, should be proportional in some way to the evolution operator $\mathbf{O}(\mathbf{s}', \mathbf{s})$. Likewise, new incident directions $(\theta_{inc}, \phi_{inc})$ should be sampled from a PDF proportional to $\mathbf{Z}(\theta_{scat}, \phi_{scat}, \theta_{inc}, \phi_{inc})$. Since PDFs are scalar functions, and that we consider the first element of the Stokes vector most important, we choose PDFs that are proportional to the (1,1) element of $\mathbf{O}(\mathbf{s}', \mathbf{s})$ and $\mathbf{Z}(\theta_{scat}, \phi_{scat}, \theta_{inc}, \phi_{inc})$.

14.2.1 Algorithm

The model algorithm proceeds as follows:

1. Begin at the cloud box exit point with a new photon. Sample a path length, Δs along the first line of sight using the PDF

$$g_0(\Delta s) = \frac{\tilde{k}\tilde{O_{11}}(\Delta s)}{1 - O_{11}(\mathbf{u_0}, \mathbf{s_0})}. \tag{14.5}$$

, where $\tilde{O_{11}}(\Delta s)$, is the piecewise exponential function that includes $O_{11}(\mathbf{s}', \mathbf{s})$ values at points where the line of sight intersects with grid surfaces. Between two such adjacent intersections, A and B, the function $\tilde{O_{11}}(\Delta s)$ is given by

$$\tilde{O_{11}}(\Delta s) = O_{11}(\Delta s_A) \exp\left(-\tilde{k}\left(\Delta s - \Delta s_A\right)\right) \tag{14.6}$$

, and

$$\tilde{k} = \frac{1}{(\Delta s_B - \Delta s_A)} \ln\left(\frac{O_{11}^A}{O_{11}^B}\right) \tag{14.7}$$

, which, for cases where the extinction matrix is diagonal, is equal to $K_{11} = (K_{11}^A + K_{11}^B)/2$. The denominator in Eq. 14.5 ensures an emission or scattering event for each photon in the initial line of sight. Eq. 14.5 is sampled by taking a random number (from the uniform distribution [0,1]), r, and solving

$$\frac{1 - \tilde{O_{11}}(\Delta s)}{1 - O_{11}(\mathbf{u_0}, \mathbf{s_0})} = r. \tag{14.8}$$

for Δs.

2. Another random number, r, is drawn to choose between emission and scattering. We first define an albedo-like quantity

$$\tilde{\omega} = 1 - \frac{K_{a1}(\mathbf{n_0}, \mathbf{s_1})}{K_{11}(\mathbf{n_0}, \mathbf{s_1})} \tag{14.9}$$

Note: we can't use the actual single-scattering albedo as this depends on the polarization state of the incident radiation. If $r > \tilde{\omega}$, then the event is considered to be emission, the reversed ray tracing is terminated, and the Stokes vector contribution of the ith photon is

$$\mathbf{I^i}(\mathbf{n}, \mathbf{s_0}) = \frac{\mathbf{O}(\mathbf{s_1}, \mathbf{s_0})\mathbf{K_a}(\mathbf{n_0}, \mathbf{s_1})I_b(T, \mathbf{s_i})}{g_0(\Delta s)\,(1 - \tilde{\omega})} \tag{14.10}$$

, where the index i signifies photon number. Return to step 1.

Otherwise, if $r \leq \tilde{\omega}$ we have a scattering event.

3. At the scattering point sample a new incident direction $(\theta_{inc}, \phi_{inc})$ according to

$$g(\theta_{inc}, \phi_{inc}) = \frac{Z_{11}(\theta_{scat}, \phi_{scat}, \theta_{inc}, \phi_{inc})\sin(\theta_{inc})}{K_{11}(\theta_{scat}, \phi_{scat}) - K_{a1}(\theta_{scat}, \phi_{scat})} \tag{14.11}$$

, which is sampled by the rejection method as described in *Liu* [2001].

Calculate the matrix

$$\mathbf{Q_k} = \mathbf{Q_{k-1}}\mathbf{q_k} \tag{14.12}$$

, where

$$\mathbf{q_k} = \frac{\sin(\theta_{inc})_k\mathbf{O}(\mathbf{s_k}, \mathbf{s_{k-1}})\mathbf{Z}(\mathbf{n_{k-1}}, \mathbf{n_k})}{g\,(\Delta s)\,g(\theta_{inc}, \phi_{inc})\tilde{\omega}}, \tag{14.13}$$

and $\mathbf{Q_0} = 1$. The index k represents the scattering order.

4. Choose a path length along the new direction according to

$$g(\Delta s) = \tilde{k}\tilde{O_{11}}(\Delta s) \tag{14.14}$$

This is sampled by taking a random number and solving

$$\tilde{O_{11}}(\Delta s) = r. \tag{14.15}$$

for Δs. If $r < O_{11}(\mathbf{u_k}, \mathbf{s_k})$, where $\mathbf{u}_k$ is the boundary of the scattering domain in the current line of sight, the photon leaves the scattering domain, and the contribution for photon i is

$$\mathbf{I^i}(\mathbf{n}, \mathbf{s_0}) = \frac{\mathbf{Q_k}\mathbf{O}(\mathbf{u_k}, \mathbf{s_k})\mathbf{I}(\mathbf{n_k}, \mathbf{u_k})}{O_{11}(\mathbf{u_k}, \mathbf{s_k})} \tag{14.16}$$

, where $\mathbf{I}(\mathbf{n_k}, \mathbf{u_k})$ is the incoming radiance at $\mathbf{u}_k$. This is calculated with the standard ARTS clear-sky routine. Return to step 1.

Otherwise, if the sampled path length keeps the path within the scattering domain...

5. As in step 2, calculate $\tilde{\omega}$ at the new point, $\mathbf{s_{k+1}}$, and draw a uniform random deviate, r.

   If $r > \tilde{\omega}$, then the event is considered to be emission, the reversed ray tracing is terminated, the Stokes vector contribution is

   $$\mathbf{I^i}(\mathbf{n}, \mathbf{s_0}) = \frac{\mathbf{Q_k O}(\mathbf{s_{k+1}}, \mathbf{s_k}) \mathbf{K_a}(\mathbf{n_k}, \mathbf{s_{k+1}}) I_b(T, \mathbf{s_{k+1}})}{g\,(\Delta s)\,(1 - \tilde{\omega})} \tag{14.17}$$

   , and we return to step 1.

   Otherwise, if $r \leq \tilde{\omega}$ we have a scattering event and we return to step 3.

6. Once the prescribed number, N, of photon contributions, $\mathbf{I^i}(\mathbf{n}, \mathbf{s_0})$, have been calculated, the cloud box exit Stokes vector is given by

   $$\mathbf{I}(\mathbf{n}, \mathbf{s_0}) = \mathbf{O}(\mathbf{u_0}, \mathbf{s_0}) \mathbf{I}(\mathbf{n}, \mathbf{u_0}) + \langle \mathbf{I^i}(\mathbf{n}, \mathbf{s_0}) \rangle. \tag{14.18}$$

   , with an estimated error for each Stokes index, j, of

   $$\delta I_j = \sqrt{\frac{\langle I_j^2 \rangle - \langle I_j \rangle^2}{N}}. \tag{14.19}$$

When simulating an MLS measurement, an extra clear sky RT calculation is performed from the cloud box exit to the sensor, with the Monte Carlo result from Eq. 14.18 taken as the radiative background.

14.3 Implementation in ARTS: ScatteringMonteCarlo

14.4 Future Plans

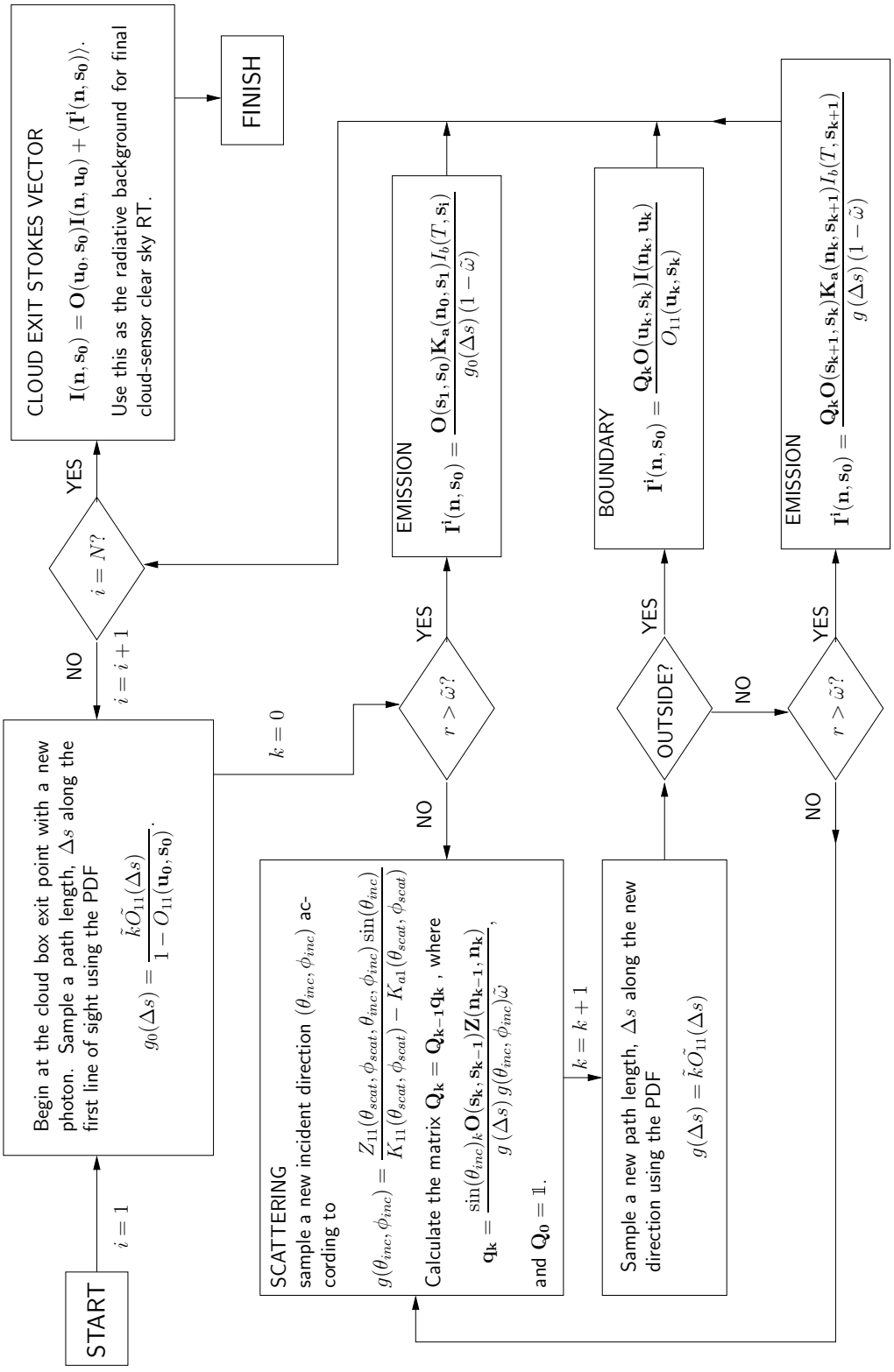

Figure 14.1: Flowchart illustrating ScatteringMonteCarlo algorithm

Part III

Implementation Issues

Chapter 15

The art of developing ARTS

The aim of this section is to describe how the program is organized and to give detailed instructions how to make extensions. That means, it is addressed to the ARTS developers, not the users. If you only want to use ARTS, you should not need to read it. **But if you want to make changes or additions, you should definitely read this carefully, since it can safe you a lot of work to understand how things are organized.**

15.1 Organization

ARTS is written in C++ with the help of the GNU development tools (Autoconf, Automake, etc.). It is organized in a similar manner as most GNU packages. The top-level ARTS directory is either called `arts` or `arts-x.y`, where x.y is the release number. It contains various sub-directories, notably `doc` for documentation, `src` for the C++ source code, `ami` for the MATLAB interface, and `aii` for the IDL interface. The document that you are reading right now, the ARTS User Guide, is located in `doc/uguide`.

There are two different versions of the ARTS package: The developers version and the end-user version. Both contain the complete source code, the only difference is that the developers version also includes the CVS housekeeping data. If you want to join in the ARTS development (which we of course encourage you to do), you should write an email to the authors to obtain access to the developers version, which makes it easier to merge your changes with the 'official' ARTS program. Furthermore, for serious development work you need a computer running Unix, the GNU development tools, LaTeX, and the Doxygen program. All this is freely and easily available on the Internet, and, what is more, all these tools are included in the standard linux distributions like Suse and Redhat.

The end-user version contains everything that you need in order to compile and install ARTS in a fairly automatic manner. The only thing you should need is an ANSI-

History	
020425	Stefan Buehler: Put this part back in the AUG. Updated.
011005	Stefan Buehler: Fixed TeX warnings, updated.
000728	Stefan Buehler: Added stuff about build system and howto cut a release.
000615	Created by Stefan Buehler. For now, this is basically the former content of the file `notes.txt`.

C++ compiler and the standard Unix `make` utility. Please see files `arts/README` and `arts/INSTALL` for installation instructions. We are developing with the GNU C++ compiler, no other compilers have been tried so far.

15.2 The ARTS build system

As mentioned above, GNU tools are used to construct the ARTS build system. A good introduction to the GNU build system can be found in:

`http://www.amath.washington.edu/ lf/tutorials/autoconf/`

Using these tools makes a lot of things very easy, but also some things slightly more complicated.

The most important thing to keep in mind is that an ARTS release is not just a copy of the ARTS development tree. Instead there is a special make target 'dist' that you can use to cut a release. How this is done in detail is described in Section 15.5.4. Mostly, the GNU tools are smart enough to figure out automatically what should go into the release. However, this can be controlled by editing the `Makefile.am` files which can be found in almost all directories.

The support for documentation other than info and man pages is not very good in the GNU system, so we had to use some tricks to make sure that the Doxygen automatic documentation and the User Guide work as they should.

15.2.1 Configure options

Here are some interesting options for `configure`/`autogen.sh`:

`--disable-debug`: Removes '-g' from the compiler flags and includes `#define NDEBUG 1` in `config.h`. The central switch to turn off all debugging features (index range checking for vectors, the trace facility, assertions,...). In maintainer mode, debugging is enabled by default.

`--disable-maintainer-mode`: Disables certain developer-centric features such as generating the user guide and the doxygen html documentation. The maintainer mode is enabled by default when using `autogen.sh` to configure arts.

`--disable-more-warnings`: Compile without `-Werror`. More warnings are enabled by default in maintainer mode except for compilers which are known to throw warnings.

`--disable-optimize`: Disable optimizations. Certain variables or functions might not be visible to the debugger when optimizations are enabled. The compiler can decide to not allocate memory for certain variables or inline whole functions for speed. This can lead to strange behaviour while debugging the program. Note: If you use the Intel C++ Compiler, disabling optimizations also turns off OpenMP parallization.

`--disable-vectorize`: Disables the generation of multi-threaded code. Configure tries to detect if the compiler supports OpenMP and enables it by default.

15.2.2 Adding directories or files

If you add directories or just files, you have to make sure that they also go into the distribution. In some cases (e.g., program source code files) this is done automatically. But if you add any other kind of file, for example a data or a documentation file, you have to edit the `Makefile.am` file in that directory to make sure that your stuff goes into the distribution. It is a good idea to always check the release in order to see if the things you added are really there.

15.3 Conventions

Here are some general rules for ARTS programming:

15.3.1 Numeric types

Never use `float` or `double` explicitly, use the type `Numeric` instead. This is set by `configure` (to `double` by default). In the same way, use `Index` for all integers. It can take on positive or negative values and defaults to `long`. To change the default types, run `configure` with the options `--with-index-type` or `--with-numeric-type`:

```
./configure --with-index-type=int --with-numeric-type=float
```

Note that changing the numeric type to a lower precision type than double might have unforseen impacts on the numerical precision and could lead to wrong results. In a similar way, reducing the index type can make it impossible to handle larger Vectors, Matrices or Tensors. The maximum range of the index type determines the maximum number of elements the container types can handle.

15.3.2 Container types

Use `Vector` and `Matrix` for mathematical vectors and matrices (with elements of type `Numeric`). Use `Array<something>` to create an array of `something`s. Commonly used Arrays have been predefined, they have names like `ArrayOfString`, `ArrayOfMatrix`, and so forth.

15.3.3 Terminology

Calculations are carried out in the so called workspace (WS), on workspace variables (WSVs). A WSV is for example the variable containing the absorption coefficients. The WSVs are manipulated by workspace methods (WSMs). The WSMs to use are specified in the controlfile in the same order in which they will be executed.

15.3.4 Global variables

Are not visible by default. To use them you have to declare them like this:

```
extern const Numeric PI;
```

which will make the global constant PI=3.14... available. Other important globals are:

`full_name`	Full name of the program, including version.
`parameters`	All command line parameters.
`basename`	Used to construct output file names.
`out_path`	Output path.
`messages`	Controls the verbosity level.
`wsv_data`	WSV lookup data.
`wsv_group_names`	Lookup table for the names of *types* of WSVs.
`WsvMap`	The map associated with `wsv_data`.
`md_data`	WSM lookup data.
`MdMap`	The map associated with `md_data`.
`workspace`	The workspace itself.
`species_data`	Lookup information for spectroscopic species.
`SpeciesMap`	The map associated with `species_data`.

The only exception from this rule are the output streams `out0` to `out3`, which are visible by default.

15.3.5 Files

Always use the `open_output_file` and `open_input_file` functions to open files. This switches on exceptions, so that any error occurring later on with this file will result in an exception. (Currently not really implemented in the GNU compiler, but please use it anyway.)

15.3.6 Version numbers

The package version number is set in file `configure.in` in the top level ARTS directory. Always increase this when you do a CVS commit, even for small changes. in such cases increase the last digit by one. If you make a new distribution, increase the middle digit by one and omit the last digit. If you make a bug-fix distribution, you can add the last digit to indicate this.

15.3.7 Header files

The global header file `arts.h` *must* be included by every file. Apart from that you have to see yourself what header files you need. If you use functions from the C or C++ standard library, you have to also include the appropriate header file.

15.3.8 Documentation

Doxygen is used to generate automatic source code documentation. See

```
http://www.stack.nl/ dimitri/doxygen/
```

for information. There is a complete User manual there. At the moment we only generate the output as HTML, although latex, man-page, and rtf format is also possible. The HTML

version is particularly useful for source code browsing, since it includes the complete source code! You should add Doxygen headers to the following:

1. Files
2. Classes (Including all private and public members)
3. Functions
4. Global Variables

The documentation headers are comment blocks that look like the examples below. They should be put above the *definition* of a function, i.e., in the `.cc` file. Some functions are defined in the `.h` file (e.g., inline member functions). In that case the comment can be put in the `.h` file.

There is an Emacs package (Doxymacs) that makes the insertion of documentation headers particularly easy. You can find documentation of this on the Doxymacs webpage: `http://doxymacs.sourceforge.net/`. To use it for ARTS (provided you have it), put the following in your Emacs initialization file:

```
(require 'doxymacs)

(setq doxymacs-doxygen-style "Qt")

(defun my-doxymacs-font-lock-hook ()
  (if (or (eq major-mode 'c-mode) (eq major-mode 'c++-mode))
      (progn
        (doxymacs-font-lock)
        (doxymacs-mode))))

(add-hook 'font-lock-mode-hook 'my-doxymacs-font-lock-hook)

(setq doxymacs-doxygen-root "../doc/doxygen/html/")
(setq doxymacs-doxygen-tags "../doc/doxygen/arts.tag")
```

The only really important lines are the first two, where the second line is the one selecting the style of documentation. The next block just turns on syntax highlighting for the Doxygen headers, which looks nice. The last two lines are needed if you want to use the tag lookup features (see Doxymacs documentation if you want to find out what this is). The package allows you to automatically insert headers. The standard key-bindings are:

`C-c d ?`	look up documentation for the symbol under the point.
`C-c d r`	rescan your Doxygen tags file.
`C-c d f`	insert a Doxygen comment for the next function.
`C-c d i`	insert a Doxygen comment for the current file.
`C-c d ;`	insert a Doxygen comment for a member variable on the current line (like M-;).
`C-c d m`	insert a blank multi-line Doxygen comment.
`C-c d s`	insert a blank single-line Doxygen comment.
`C-c d @`	insert grouping comments around the current region.

You can call the macros also by name, e.g., `doxymacs-insert-file-comment`.

File comment

Generated by `doxymacs-insert-file-comment`.

```
/*!
\file   dummy.cc
\author Stefan Buehler <sbuehler (at) irv.se>
\date   Thu Apr 25 15:58:50 2002

\brief  A dummy file.

 This file has no purpose at all,
 it just servers as an example...
*/
```

Function comment

Generated by `doxymacs-insert-function-comment`. If arguments are modified by the function you should add 'Output:' after the
`param` command, just like for the parameter `a` in the example below. If a parameter is both input and output, you should say 'Output and Input:'. The documentation for each parameter should start with a capital letter and end with a period, like in the example below.

Author and date tags are not inserted by default, since they would be overkill if you have many small functions. However, you should include them for important functions.

```
//! A dummy function.
/*!
 This function has no purpose at all,
 it just serves as an example...

\param  a Output: This parameter is modified by the
          function.
\param  b This is the other parameter.
\return   Dummy value computed from a and b.
*/
int dummy(int& a, int b);
```

Generic multi-line comment

Generated by `doxymacs-insert-blank-multiline-comment`.

```
//! A dummy comment.
/*!
 Some more elaborate description about this variable,
 class, or whatever.
*/
```

Generic single-line comment

Generated by `doxymacs-insert-blank-singleline-comment`.

```
//! Short comment here.
```

15.4 Extending ARTS

15.4.1 How to add a workspace variable

You should read Sectionsec:agendas:wsvs to understand what workspace variables are. Here is just the practical description how a new variable can be added.

1. Create a record entry in file `workspace.cc`. (Just add another one of the `wsv_data.push_back` blocks.) Take the already existing entries as templates. The ARTS concept works best if WSVs are only of a rather limited number of different types, so that generic WSMs can be used extensively, for example for IO.

   The name must be *exactly* like you use it in the source code, because this is used to generate interface functions.

   Make sure that the documentation string you give explains the variable and its purpose well. **In particular, state the dimensions (in the case of matrices) and the units!** This string is used for the online documentation. Please take some time to write it carefully. Use the template at the beginning of function `define_wsv_data()` in file `workspace.cc` as a guideline.

2. That's it!

15.4.2 How to add a workspace variable group

You should read Sectionsec:agendas:wsvs to understand what workspace variable groups are. Here is just the practical description how a new group can be added.

1. Add a `wsv_group_names.push_back("your_type")` function to the function `define_wsv_group_names()` in `groups.cc`. The name must be *exactly* like you use it in the source code, because this is used to generate interface functions.

2. XML reading/writing routines are mandatory for each workspace variable group. Two steps are necessary to add xml support for the new group:

   (a) Implement an `xml_read_from_stream` and `xml_write_to_stream` function. Depending on the type of the group the implementation goes into one of the three files `xml_io_basic_types.cc`, `xml_io_compound_types.cc`, or `xml_io_array_types.cc`. Basic types are for example Index or Numeric. Compound types are structures and classes. And array types are arrays of basic or compound types. Also add the function declaration in the corresponding `.h` file.

   (b) Add an explicit instantiation for `xml_read_from_file<GROUP>` and `xml_write_to_file<GROUP>` to `xml_io_instantiation.h`.

3. If your new group does not implement the output operator (`operator<<`), you have to add an explicit implementation of the `Print` function in `m_general.h` and `m_general.cc`.

4. That's it! (But as stated above, use this feature wisely)

15.4.3 How to add a workspace method

You should read Sectionsec:agendas:wsms to understand what workspace methods are. Here is just the practical description how a new method can be added.

1. Create an entry in the function `define_md_data` in file `methods.cc`. (Make a copy of an existing entry (one of the `md_data.push_back(...)` blocks) and edit it to fit your new method.) Don't forget the documentation string! Please refer to the example at the beginning of the file to see how to format it.

2. Run: `make`.

3. Look in `auto_md.h`. There is a new function prototype

   ```
   void <YourNewMethod>(...)
   ```

4. Add your function to one of the `.cc` files which contain method functions. Such files must have names starting with `m_`. (See separate HowTo if you want to create a new source file.) The header of your function must be compatible with the prototype in `auto_md.h`.

5. Check that everything looks nice by running

   ```
   arts -d YourNewMethod
   ```

   If necessary, change the documentation string.

6. Thats it!

15.4.4 How to add a source code file

1. Create your file. Names of files containing workspace methods should start with `m_`.

2. You have to register your file in the file `src/Makefile.am`. This file states which source files are needed for arts. Should be self-explanatory where you have to add your file. The above goes for source (`.cc`) and header (`.h`) files likewise.

3. Then go to the top level arts directory and run: `autogen.sh`.

4. Go to `src` and run: `cvs add <my_file>` to make your file known to CVS.

15.4.5 How to add a test case

1. Create a new subdirectory in `tests/`. If your test is closely related to another test case you can skip this step and instead add it to one of the existing subdirectories.
2. Create your own test controlfile. The filename should start with `Test` followed by the name of the subdirectory it is located in, e.g. `tests/DOIT/TestDOIT.arts`. If the subdirectory contains more than one test controlfile append a short descriptive text to the end of the filename like `tests/MonteCarlo/TestMonteCarloGaussian.arts`.
3. Copy all required input files into the subdirectory.
4. Add the necessary checks for your testcase to `tests/testall.py`.
5. In `tests/Makefile.am` add the name of your test to the variable `check` and the subdirectory to the variable `TESTDIRS`.

15.5 SVN issues

The arts project is controlled by Subversion. This section describes some basic SVN commands. For more information see the extensive SVN documentation:

`http://svnbook.red-bean.com/`

15.5.1 How to check out arts

1. Go to a temporary directory.
2. Run: `svn co https://www.sat.ltu.se/svn/rt/arts/trunk arts`

15.5.2 How to update (if you already have a copy)

1. Go to the top ARTS directory (called simply `arts`).
2. Run: `svn update`

   IMPORTANT! Always update, before you start to make changes to the program, especially after a longer pause. If you edit an outdated copy, it will be a lot more work to bring your changes into the current copy of the program.

15.5.3 How to commit your changes

1. You should make sure that the program compiles and runs without obvious errors before you commit.
2. If you have created a new source file, make it known to SVN by running the command `svn add <my_file>` in the directory where the file resides.

   In general, when you run `svn update`, it will warn you about any files it doesn't know by marking them with a `?`. Files that are created during the compilation process,

but should not be part of the package are ignored by SVN. You can view the list of ignored files for the current directory with `svn propget svn:ignore ..` The list can be edited with `svn propedit svn:ignore ..`

3. Have you added the documentation for your new features?

4. Increase the subversion number in file `configure.in` in the top level ARTS directory.

5. Open the file `ChangeLog` in the top level ARTS directory with your favorite editor.

   With Emacs, you can very easily add an entry by typing either

   ```
   M-x add-change-log-entry
   ```

   or `C-x 4 a`.

   Specify the new version number and describe your changes.

   These keystrokes work also while you are editing some other file in Emacs. Thus it is best to write your ChangeLog entry already while you work on a file. Whenever you make a change to a file, there should be a ChangeLog Entry!

6. Make sure that you have saved all your files. Go to the top level ARTS directory and run: `svn commit`.

7. This will pop up an editor. Use the mouse to cut and paste the Change-Log message also to this editor window. Safe the file and exit the editor. If you made changes in different directories, another editor will pop up, already containing your message. Save again and exit. Do this until no more editors come up. (Note: This works well if you set

   ```
   export EDITOR=xedit
   ```

   in your shell startup file.

   With smart editors there can be problems, because they might refuse to safe your file if you haven't made changes to it. With xedit you just have to push the save button twice to override.

8. Tell the other developers about it. The best way to do this is to send an email to `arts-dev@www.sat.ltu.se`.

15.5.4 How to cut a release

1. Change the release number in the file `configure.in` in the top-level ARTS directory. (The line that you have to change is the one with `AM_INIT_AUTOMAKE`.) Omit the subversion number (last digit).

2. Commit your changes (see other howto).

3. In the top-level ARTS directory, run `autogen.sh`.

4. In the top-level ARTS directory, run `make distcheck`. This will not only cut the release, but also immediately try to build it, to see if it works. Unless you are on a very fast machine, this may take a while. Maybe you should go and have a cup of coffee.

5. If all goes well, you can find the release inside the top-level ARTS directory as a file `arts-x.y.tar.gz`, where x.y is the release number.

6. Check the release carefully by trying to build and install the program.

15.5.5 How to move your arts working directory

In general it is no problem to move your working directory. The only thing to consider is that the autotools write hardwired paths into the generated Makefiles. Therefore you have to run `autogen.sh` working directory.

15.6 Debugging (use of assert)

This section draws heavily on the GNU tools manual of Eleftherios Gkioulekas:

`http://www.amath.washington.edu/~lf/tutorials/autoconf/`

The idea behind assert is simple. Suppose that at a certain point in your code, you expect two variables to be equal. If this expectation is a precondition that must be satisfied in order for the subsequent code to execute correctly, you must assert it with a statement like this:

```
assert(var1 == var2);
```

In general assert takes as argument a boolean expression. If the boolean expression is true, execution continues. Otherwise the `abort` system call is invoked and the program execution is stopped. If a bug prevents the precondition from being true, then you can trace the bug at the point where the precondition breaks down instead of further down in execution or not at all. The `assert` call is implemented as a C preprocessor macro, so it can be enabled or disabled at will.

In ARTS, you don't have to do this manually, as long as your source file includes `arts.h` either directly or indirectly. Instead, assertions are turned on and off with the global NDEBUG preprocessor macro, which is set or unset automatically by the `configure` script. The relevant `configure` options are `--enable-debug` and `--disable-debug`. Assertions are also turned on automatically by the `--enable-maintainer-mode` option. (And this again is set automatically if you run the `autogen.sh` script.)

If your program is stopped by an assertion failure, then the first thing you should do is to find out where the error happens. To do this, run the program under the GDB debugger. First invoke the debugger:

```
gdb arts
```

You have to give the full path to the ARTS executable. Then set a breakpoint at the assertion failure:

```
(gdb) break __assert_fail
```

(Note the two leading underscores!) Now run the program:

```
(gdb) run
```

Instead of just exiting, under the debugger the program will be paused when the assertion fails, and you will get back the debugger prompt. Now type:

```
(gdb) where
```

to see where the assertion failure happened. You can use the `print` command to look at the contents of variables and you can use the `up` and `down` commands to navigate the stack. For more information, see the GDB documentation or type `help` at the prompt of GDB.

For ARTS, the assertion failures mostly happen inside the Tensor / Matrix / Vector package (usually because you triggered a range check error, i.e., you tried to read or write beyond array bounds). In this case the `up` command of GDB is particularly useful. If you give this a couple of times you will finally end up in the part of your code that caused the error.

Recommendation: In Emacs there is a special GDB mode. With this you can very conveniently step through your code.

Chapter 16

The workspace

FIXME: This is a construction site. Please don't read!

This chapter deals with the main components of ARTS: *Workspace variables* (WSVs) and *workspace methods* (WSMs). Furthermore, it explains the use of agendas, a special group of WSVs.

16.1 Implementation files

The two most important files are:

- `workspace.cc`:
  Definition and documentation of WSVs.
- `methods.cc`:
  Definition and documentation of WSMs. The implementations of WSMs reside in files named `m_something.cc`.
- `agendas.cc`:
  Definition and documentation of agendas.

It is very likely that you will have to edit these. Less likely, but possibly, you also have to edit:

- `groups.cc`:
  Definition of WSV groups.

When ARTS is built, a number of source code files are generated automatically. They are listed here in the order in which they are generated:

- `auto_wsv_groups.h`:
  Generated from `groups.cc`.

History
020605 Created by Stefan Buehler.

- auto_wsv.h, auto_wsv_pointers.cc:
  Generated from auto_wsv_groups.h and workspace.cc.

- auto_md.h, auto_md.cc:
  Generated from auto_wsv_groups.h, auto_wsv.h, agendas.cc, and methods.cc.

This is achieved by a set of simple C++ programs:

- make_auto_wsv_groups_h.cc
- make_auto_wsv_h.cc
- make_auto_wsv_pointers_cc.cc
- make_auto_md_h.cc
- make_auto_md_cc.cc

The meaning of the names should be self-explanatory. There is one program for each file to be generated. The generation of the auto_ files happens automatically when you do a make. Therefore, never edit any of these files.

Next, there are some files that contain the internal implementation of WSVs and WSMs. These are:

- wsv_aux.h, wsv_aux.cc, workspace_aux.cc:
  Implementation of class WsvRecord, which stores the lookup information for one WSV, plus auxiliary stuff for the workspace.

- methods.h, methods_aux.cc:
  Implementation of class MdRecord, which stores the lookup information for one WSM.

Finally, there are some files that contain the internal implementation of agendas. These are:

- agenda_class.h, agenda_class.cc:
  Implementation of class MRecord, which stores runtime information for one WSM, and class Agenda, which stores an agenda.

- agenda_record.h, agenda_record.cc:
  Implementation of class AgRecord, which is used to store agenda lookup information.

As mentioned above, you will not have to modify any of the implementation files, they are listed here just for reference. Normally, you only have to modify workspace.cc, methods.cc, and agendas.cc.

16.2 Workspace Variables or WSVs

All important variables in ARTS are WSVs. This means that they can be manipulated by a list of WSMs, which is specified in the ARTS controlfile. There exists a predefined list of possible WSVs. This list defines the *workspace*. One can think of each WSV as a 'slot' in the workspace: The WSV can be either *set*, or *unset*. Set means that the WSV has a well-defined content, unset means that it has no well-defined content. At the start of an ARTS job all WSVs are unset.

WSVs are defined in the file `workspace.cc`. A typical definition looks like this:

```
wsv_data.push_back
  (WsvRecord
   ( NAME( "f_grid" ),
     DESCRIPTION
     (
      "The frequency grid for monochromatic pencil beam\n"
      "calculations.\n"
      "\n"
      "Usage:      Set by the user.\n"
      "\n"
      "Unit:       Hz"
      ),
     GROUP( "Vector" )));
```

All WSV definitions have the same three elements:

1. The *name*, exactly the same name has to be used in the code.
2. The *description*, which is normally much longer than in the example here. It must fully describe the WSV, its purpose, and its normal usage. See file `workspace.cc` for instructions how to write the documentation.
3. The *group* to which the WSV belongs. You can think of a group as something similar to a C++ data type. The WSV in the example belongs to the group `Vector`. The allowed groups are defined in file `groups.cc`. Note that you have to add an underscore to the group name.

See Section 15.4 for explicit instructions how to add a new WSV to ARTS.

16.3 Workspace Methods or WSMs

WSMs manipulate WSVs to produce other WSVs. There are three kinds of WSMs:

1. Specific WSMs.
2. Generic WSMs.
3. Agenda WSMs.

As in the case of WSVs, there is a central place in ARTS where information on the available WSMs is stored. This place is the file `methods.cc`. It contains a record for each WSM. Here is an example:

```
md_data.push_back
  ( MdRecord
    ( NAME("elsLorentz"),
      DESCRIPTION
      (
       "The Lorentz lineshape.\n"
       "\n"
       "This computes the simple Lorentz lineshape as:\n"
       "\n"
       "els[i] = 1/PI * ls_gamma /\n"
       "         ( (els_f_grid[i])^2 + ls_gamma^2 )\n"
       "\n"
       "Note that the frequency grid els_f_grid must hold\n"
       "offset frequencies from line center. Hence, the\n"
       "line center frequency is not needed as input.\n"
       "\n"
       "Output:\n"
       "   els        : The lineshape function [1/Hz]\n"
       "\n"
       "Input:\n"
       "   ls_gamma   : Line width [Hz].\n"
       "   els_f_grid : Frequency grid [Hz]."
      ),
      OUT( "els" ),
      GOUT(),
      GOUT_TYPE(),
      IN( "ls_gamma", "els_f_grid" ),
      GIN(),
      GIN_TYPE(),
      GIN_DEFAULT(),
    ));
```

All WSM definitions have the same elements:

1. The *NAME*, exactly as in the code.
2. The *DESCRIPTION*. This must fully describe the WSM, its purpose, and its normal usage. See file `methods.cc` for instructions how to write the documentation.
3. The *OUT*. This is a list of WSV names. All these WSVs are set by this WSM.
4. The *GOUT*. This is a list descriptive names for the generic outputs.
5. The *GOUT_TYPE*. This is a list of WSV group names. This defines the group to which output arguments must belong (see below).
6. The *IN*. This is a list of WSV names. All these WSVs are required as input by this WSM. This means they must have been set before.
7. The *GIN*, a list of descriptive names for the generic inputs.
8. The *GIN_TYPE*. This is a list of WSV group names This defines the group to which input arguments must belong.

9. The *GIN_DEFAULT*, a list of default values for the generic inputs. `NODEF` means that the generic input has no default and the user has to set it in the control file.

16.3.1 Specific WSMs

For this type of WSM the output and input is fixed. Fields `GIN` and `GOUT` are empty. The example above belongs in this category. It sets the WSV `els`, using the WSVs `ls_gamma` and `els_f_grid` as inputs. What the function actually does is to compute a Lorentzian lineshape function with width `ls_gamma`, for the frequencies given in the grid `els_f_grid`. (The line center is at frequency 0.) The result is then stored in the output WSV `els`.

To call this method in the controlfile, you just have to write `elsLorentz`.

16.3.2 Generic WSMs

This class of WSM is more powerful, because it can be applied to any WSV that belongs to the right group. A good example is:

```
md_data_raw.push_back
  ( MdRecord
    ( NAME("VectorSetConstant"),
      DESCRIPTION
      (
       "Creates a workspace vector and sets all elements of the \n"
       "vector to the specified value. The length of the vector is \n"
       "determined by the variable *nelem*. \n"
      ),
      AUTHORS( "Patrick Eriksson" ),
      OUT(),
      GOUT( "vector" ),
      GOUT_TYPE( "Vector" ),
      IN( "nelem" ),
      GIN(          "value"   ),
      GIN_TYPE(     "Numeric" ),
      GIN_DEFAULT( NODEF )
    ));
```

As you probably have guessed, this WSM resizes the output vector to have `nelem` elements and sets all elements to the given `value`. You would use it as follows:

```
IndexSet(nelem, 10)
VectorCreate(myvector)
VectorSetConstant(myvector, nelem, 0)
```

This would create the WSV `myvector` and then fill it with 10 elements set to 1. Note that output arguments always come first, input arguments last. Try `arts -d VectorSetConstant` to get more information on this method. (See section 2.4 for information on command line switches.)

For basic types it is allowed to pass values instead of variables directly to the WSM. In that case, the above example would look like this:

```
VectorCreate(myvector)
VectorSetConstant(myvector, 10, 0)
```

16.3.3 Agenda WSMs

16.4 Agendas

16.4.1 Introduction

Agendas are a special incarnation of a WSM. At runtime an arbitrary number of WSMs can be added to an agenda. On invocation, the agenda will execute its methods one after the other. The inputs and outputs defined for the agenda must be satisfied by the invoked WSMs. E.g., if an agenda has `f_grid` in its list of output WSVs, a WSM which generates `f_grid` must be added to the agenda in the control file.

Agendas run their methods in a separate scope. Although WSMs invoked by an agenda have full access to all workspace variables, only the WSVs defined as output of the agenda will keep their values after the agenda execution. All other WSVs retain the values from before the agenda run.

Even though it is possible to execute agenda directly from the control file with the `AgendaExecute` method, the more common and intended use case is the internal invocation by other WSMs. This adds a considerable amount of flexibility to arts. The `RteStd` method for example calculates (besides other components) the emission term. Without the means of an agenda, it would only be possible to use always the same method for the emission calculation. By the use of an agenda the user can choose between different methods to calculate the emission and plug them into the emission agenda in the control file:

```
AgendaSet( emission_agenda ){
  emissionPlanck
}
```

Part IV

Mathematical functions

Chapter 17

Vectors, matrices, tensors, and arrays

This section describes how vectors and matrices are implemented in ARTS and how they are used. Furthermore it describes how arrays of arbitrary type can be constructed and used.

17.1 Implementation files

The `Matrix` and `Vector` classes described below reside in the files:

- `matpackI.h`
- `make_vector.h`
- `matpackI.cc`
- `make_vector.cc`

Tensors of order 3 to 7 reside in the files:

- `matpackIII.h`
- `matpackIV.h`
- `matpackV.h`
- `matpackVI.h`
- `matpackVII.h`

The template class `Array` (also described below) is implemented in the files:

- `array.h`

History	
030807	Sparse added by Mattias Ekström.
030109	Documentation for using jokers without Range added by Stefan Buehler.
020516	Tensors added by Stefan Buehler.
011018	Created and written by Stefan Buehler.

- `make_array.h`

The `Sparse` class is described in the file:

- `matpackII.h`

The file `test_matpack.cc` contains test cases and usage examples. For Sparse there is a separate test file, `test_sparse.cc`.

17.2 Vectors

The class `Vector` implements the mathematical concept of a vector. (Surprise, surprise.) This means that:

- A Vector contains a list of floating point values of type `Numeric`.
- A Vector can be multiplied with another Vector (scalar product), or with a Matrix.
- Sub-ranges of a Vector can easily be accessed, and used as if they were Vectors.
- Resizing a Vector is expensive and should be avoided.

17.2.1 Constructing a Vector

You can construct an object of class Vector in any of these ways:

```
Vector a;          // Create empty Vector.
Vector b(3);       // Create Vector of length 3, if
                   // created like this it will contain
                   // arbitrary values.
Vector c(3,0.0);   // Create Vector of length 3, and
                   // fill it with 0.

Vector d=c;        // Make d a copy of c.

Vector e(1,5,1);   // 1, 2, 3, 4, 5
Vector f(1,5,.5);  // 1, 1.5, 2, 2.5, 3
Vector g(5,5,-1);  // 5, 4, 3, 2, 1
```

The last three examples all use the same constructor, which takes the three arguments ‘start’, ‘extent’, and ‘stride’. It will create a Vector containing ‘extent’ elements, starting with ‘start’, with a step of ‘stride’.

There also exists a special sub-class of Vector that can be initialized explicitly. This must be a special class in order to avoid ambiguities with the standard constructors. Usage:

```
MakeVector a(1.0,2.0,3.0);  // Creates a vector of length 3
                            // containing the values
                            // 1.0, 2.0, and 3.0.
```

You can use MakeVectors just like Vectors, except that the constructors are different. Otherwise you can mix them freely with Vectors.

17.2.2 VectorViews

An object of class `VectorView` is, like the name says, just another view on an existing Vector. It does not have its own data. This has the important consequence that it cannot be resized, since that would mess up the original Vector that the view is referring to. You can create VectorViews from Vectors using the index operator '[]', the class `Range`, and the special `joker` object. Examples:

```
MakeVector x(1,2,3,4,5,6,7);
VectorView a = x;                  // Now a refers to the
                                   // whole of x;
VectorView b = x[Range(joker)];    // Same effect.
VectorView c = x[Range(0,2)];      // Take 2 elements of x,
                                   // starting at the
                                   // beginning,
                                   // in this case: 1,2.
VectorView d = x[Range(0,3,2)];    // In this case: 1,3,5.
VectorView e = x[Range(3,joker)]; // In this case: 4,5,6,7.
```

As you can see, most useful ways to create VectorViews involve the Range class. The general constructor to this class takes three arguments, 'start', 'extent', and 'stride'. This means that you will select 'extent' elements from the Vector, starting with index 'start', with a step-width of stride. Note that indices are 0-based, so 0 refers to the first element. The last argument, 'stride', can be omitted, in that case the default of 1 is assumed. As a special case, 'extent'==`joker` means 'to the end', and calling Range with only one argument `joker` means 'all elements'.

Usually, you will not have to use VectorView explicitly, because you can use expressions like:

```
Vector a(1,5,1);                  // a = 1,2,3,4,5
Vector b = a[Range(1,3)];         // b = 2,3,4
```

However, `VectorView` and the related class `ConstVectorView` are extremely useful as the argument types of functions operating on Vectors. You should define your functions like this:

```
void silly_function(VectorView a,       // Output argument
                    ConstVectorView b   // Input argument
                                        // (read only)
                   )
{
   // Do some silly stuff with a and b.
}
```

Note that there must not be any '&' after VectorView or ConstVectorView. In other words they have to be passed by value, not by reference. This is ok, since they do not contain the actual data, so that passing by value is efficient. Passing VectorViews by reference is forbidden.

You should use these kind of arguments for all input Vectors, and also for the output if you have a function that does not resize the output Vector. This has the great advantage that you can call the function with Vector sub-ranges, e.g.,

```
Vector a(1,5,1);                    // a = 1,2,3,4,5
Vector b(3);                        // Set size of b.
silly_function(b,a[Range(0,3)]);    // Call fuction with
                                    // sub-range of a.
```

An exception to this rule are workspace methods, which use conventional argument types `const Vector&` for input and `Vector&` for output.

17.2.3 What you can do with a Vector (or VectorView)

All examples below (except for the first) assume that `a` is a Vector, MakeVector, or VectorView.

Resize (only for Vector, not for VectorView!):

```
a.resize(5);
```

This makes `a` a 5 element vector. The new Vector is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost. Appending to a Vector is not possible.

Get the number of elements:

```
cout << a.nelem();
```

Sum up all elements:

```
cout << a.sum();
```

Element access:

```
cout << a[3];   // Print 4th element.
a[0] = 3.5;     // Assign 3.5 to first element.
```

Note that we use 0-based indexing! Furthermore note that the operator '[]' can be also used with `Range`, as explained above.

Copying Vectors:

```
Vector b;
b = a;
```

In this case the size of b will be adjusted to that of a automatically. Maybe you have noticed that there is a way to formulate the example above in even shorter fashion:

```
Vector b = a;
```

The result is exactly the same. Note, though, that in this case b is *constructed* from a, not copied (see section about constructing Vectors above).

Copying in connection with views:

This one is a bit tricky. Obviously, the size of views can not be adjusted, because a view is just some selection of the underlying object. The '=' operator in this case copies the *contents*, so the sizes of the left-hand and right-hand argument must match. VectorView internally uses assertions to make sure of this. So, if you get an assertion failure one reason could be that you forgot to make the target the correct size. Here is an example:

```
b[Range(5,5,-1)] = a[Range(3,5)];  // Copy 5 elements from
                                   // a to b, reversing
                                   // the order and starting
                                   // with index 3 in a.
```

Great, isn't it?

Assigning a scalar:

```
a = 1.0;                           // Assign 1 to all elements.
```

Mathematical operators:

```
Vector a(1,3,1), b(3,1); // a = 1,2,3; b = 1,1,1
a *= 2;                  // a = 2,4,6
                         // Similarly, /=, +=, -=
a += b;                  // a = 3,5,7
                         // Similarly, -=, *=, /=
a += a;                  // a = 6,10,14
                         // So a can appear on both sides.
```

All these operate element-wise. Note, that there are no return versions of these operators (i.e., expressions like `b = a+1` are not possible). This is again for efficiency reasons. It is currently an active area of research in programming techniques how to make this kind of expression efficient. None of the available solutions works, so ARTS has to live without it.

Maximum and minimum:

```
cout << max(a);
cout << min(a);
```

Scalar product:

```
cout << a*a;
```

This is an exception to the rule not to have return versions of operators. The reason is quite obvious: The return value is only a scalar.

Arbitrary single-argument math functions:

```
Vector b(a.nelem());
transform(b,sin,a);  // b = sin(a)
transform(b,cos,b);  // b = sin(b)
                     // So b can appear on both sides.
```

The transform function operates on each element of `a` with the function you specify and puts the result in `b`. Note that the order of the arguments is swapped compared to the old function `trans` that we had in the pre-Matpack era.

17.3 Matrices

The class `Matrix` implements the mathematical concept of a matrix. (Who would have guessed this?) This means that:

- A Matrix contains floating point values of type `Numeric`.
- The values are arranged in rows and columns and can be accessed by indices. The first index is the row, the second the column. In other words, we use *row-major* order, similar to C, Matlab, and most math textbooks. Note, however, that some languages like FORTRAN and IDL use *column-major* order.
- A Matrix can be multiplied with a Vector, or with another Matrix.
- A sub-range of a Matrix in both dimensions (submatrix) can easily be accessed, and used as if it was just a normal matrix.
- Resizing a Matrix is expensive and should be avoided.

17.3.1 Constructing a Matrix

You can construct an object of class Matrix in any of these ways:

```
Matrix a;           // Create empty Matrix.
Matrix b(3,4);      // Create Matrix with 3 rows
                    // and 4 columns. When
                    // created like this it will contain
                    // arbitrary values.
Matrix c(3,4,0.0); // Similar, but
                    // fill it with 0.

Matrix d=c;         // Make d a copy of c.
```

That is all. More fancy constructors, like for Vector, do not exist for Matrix. There is also no equivalent to the `MakeVector` class.

17.3.2 MatrixViews

A `MatrixView` is a view on an existing Matrix, in the same way as a `VectorView` is a view on an existing Vector. Like a VectorView, a MatrixView cannot be resized and does not contain the actual data. A view is generated by using Ranges:

```
Matrix x(10,20);                    // Create 10x20 matrix.
MatrixView a = x;                   // Now a refers to the
                                    // whole of x;
MatrixView b = x(Range(joker),Range(joker));
                                    // Same effect.
MatrixView c = x[Range(0,2),Range(0,2)];
                                    // 2x2 sub-matrix.
```

You probably get the idea. Note that the second argument of Range gives the number of elements to take, not the index of the last element. See the section about Vectors for more examples how to use Range. You can use `joker`, and also the third argument of Range to select only every nth row, or column, or reverse the order of the rows or columns.

In analogy to the Vector case, you should use the two classes `MatrixView` and `ConstMatrixView` as function arguments. Please refer to the discussion in the Vector section for details. As in the case of VectorViews, all arguments of these types should be passed by value, not by reference. Also, similar to the Vector case, workspace methods are the exception, because they have to use the conventional `const Matrix&` or `Matrix&` as input/output arguments.

17.3.3 What you can do with a Matrix (or MatrixView)

All examples below (except for the first) assume that `a` is a Matrix or MatrixView.

Resize (only for Matrix, not for MatrixView!):

```
a.resize(5,10);
```

This makes `a` a 5x10 Matrix (5 rows, 10 columns). The new Matrix is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost.

Get the number of rows or columns:

```
cout << a.nrows();
cout << a.ncols();
```

Refer to a row or column:

```
Vector x = a(0,Range(joker));           // First row.
Vector y = a(Range(joker),a.ncols()-1); // Last column.
```

Of course, you can use more complicated Range expressions to refer to only parts of a row or column. However, the case that you want all elements of a given dimension is so much more common than the more sophisticated uses of the `Range` class, that it is worth to introduce a simplified notation for this case. Therefore, you are allowed to omit the `Range` and just write:

```
Vector x = a(0,joker);            // First row.
Vector y = a(joker,a.ncols()-1); // Last column.
```

Technically, expressions of this kind return the type `VectorView`. This means, they can be used in all cases where an object of that type is expected, for example with the function defined in Section 17.2.2:

```
silly_function(a(0,Range(joker)),
               a(1,Range(joker))); // Call silly_function
                                   // with first and
                                   // second row of a.
```

Element access:

```
cout << a(3,4); // Print that element.
a(0,0) = 3.5;   // Assign 3.5 to the top-left element
```

Note that we use 0-based indexing! Furthermore note that the operator '()' can be also used with one or two `Range` arguments, as explained above. To summarize:

- (`Index`,`Index`) returns `Numeric` (element access).
- (`Index`,`Range`) or (`Range`,`Index`) returns `VectorView` (row or column access).
- (`Range`,`Range`) returns `MatrixView` (sub-matrix access).

You may find it unlogical, that Matrix uses '()' for indexing, whereas Vector uses '[]'. However, using '[]' for Matrix is not possible, since it can have only one argument. On the other hand, using '()' for Vector element access seemed not a good idea, since that would break with the established use of '[]' for element access in C and C++.

Copying Matrices:

```
Matrix b;
b = a;
```

As in the case of Vectors, the '=' operator adjusts the size of the target automatically.

Copying in connection with views:

As in the case of Vectors, the '=' operator copies only the *contents* for views, so the dimensions must match. An attempt to justify this behavior has been made above in the Section about Vector. As for Vector, you can use '=' with complicated expressions. Here is a more elaborate example:

```
b(Range(0,3),Range(0,4)) =
   a(Range(10,3),Range(3,4,-1)); // Copy a row 10-12,
                                 // column 0-3
                                 // to b row 0-2,
                                 // column 0-3, reversing
                                 // the order of columns.
```

Note that in this case the dimensions must match exactly, as explained in the Section about Vector.

If you do not understand the use of Range here, refer to Section 17.2.2.

Assigning a scalar:

```
a = 1.0;                          // Assign 1 to all elements.
```

Mathematical operators:

You can use the operators '+=', '-=', '*=', and '/=', which operate element-vise, just as for Vector.

Maximum and minimum:

```
cout << max(a);
cout << min(a);
```

Arbitrary single-argument math functions:

The function `transform` works just like for Vector.

Transpose:

```
Matrix b = transpose(a); // Make b the transpose of a.
```

The function `transpose` creates a MatrixView, for which rows and columns are interchanged. Note, that only the way the data is accessed is changed, not the data itself. So Matrix `a` in the example above is not changed. For this reason, transposing is very efficient. You can use `transpose(a)` instead of `a` in any matrix expression practically without additional cost. (This is not strictly true, after all, the view has to be generated and passed. But that cost should be negligible except for very small matrices.)

Matrix multiplication:

```
// Matrix-Vector:
Vector b(a.nrows()), c(a.ncols());
mult(b,a,c);                              // b = a * c

// Matrix-Matrix:
Matrix d(a.nrows(),5), e(a.ncols(),5);
mult(d,a,e);                              // d = a * e
```

Note, that the result is put in the first argument, consistent with the general ARTS policy, but different from the old MTL based multiplication function. Furthermore note, that as you can see from the first example, a Vector is always considered to be a 1-column Matrix.

Important: The matrices or vectors that you give for the three arguments must not overlap, or you will get garbage. In particular, this means that

```
mult(x,y,x);              // x = y * x FORBIDDEN!!!
```

does not work. No, even worse: It works, but it gives the wrong result. The reason for this behavior is that the result is constructed in the first argument variable. If that is also an input variable it will change while it is multiplied, which will lead to a different result. There is no efficient way to detect overlap, so the only way to allow input and output arguments to be identical would be to use another internal dummy variable to store the result. However, this would be much less efficient.

Another thing: You can use transpose, of course. These two examples should obviously give the same result:

```
// Define b and c as in first example above.
mult(c,transpose(a),b);                       // c = a' * b

// Vector-Matrix:
mult(transpose(c),transpose(b),a);            // c' = b' * a
```

17.4 Tensors

ARTS has tensors with rank 3 to 7. They are called `Tensor3`, `Tensor4`, `Tensor5`, `Tensor6`, `Tensor7`, and work very much like matrices, just with more dimensions. Some properties:

- A Tensor contains floating point values of type `Numeric`.
- The *rank* of a tensor means the number of dimensions, so a `Tensor4` has 4 dimensions. Tensors of different rank are different classes. That means, the rank is fixed at compile time and cannot be changed at runtime. We will use rank and dimension as synonyms.
- The different dimensions are named:
  - Library

  - Vitrine
  - Shelf
  - Book
  - Page
  - Row
  - Column

  For example, `Tensor3 b(2,4,3)` defines a third order tensor with 2 pages, 4 rows, and 3 columns. Note that the column dimension is always last. (Incidentally, a `Matrix` behaves exactly like a second order tensor, except that it has some additional features.)

- A sub-range of a tensor in all dimensions (sub tensor) can easily be accessed, and used as if it was just a normal tensor.
- More importantly, you can easily access lower dimensional 'slices' of a tensor.
- Resizing a tensor is expensive and should be avoided.

17.4.1 Constructing a tensor

You can construct an object of a tensor class like this:

```
Tensor7 a;             // Create empty tensor of rank 7
Tensor3 b(2,4,3);      // 2 pages, 4 rows, 3 columns
Tensor3 c(2,4,3,0.0); // Similar, but
                       // fill it with 0.

Tensor3 d=c;           // Make d a copy of c.
```

17.4.2 Tensor views

Tensor views work exactly like matrix and vector views. Example:

```
Tensor4 a(10,20,5,4);
Tensor3View b = a(3,Range(1,3),joker,joker);
```

If you have read the previous sections carefully, it should be clear what this expression does.

This is what is meant by slicing: You can easily create a view of a tensor that picks out an object of lower dimension. Note that you can use either an `Index` or a `Range` argument[1] for any of the dimensions. The dimensionality of the result will adjust accordingly, as in the example above.

Everything that was said about matrix and vector views holds also here. In particular, please always use views as function arguments.

[1] Using just `joker` is equivalent to using `Range(joker)`, as explained in Section 17.3.

17.4.3 What you can do with a tensor (or tensor view)

All examples below (except for the first) assume that `a` is a Tensor7 or Tensor7View.

Resize (only for tensors, not for views):

```
a.resize(5, 10, 4, 5, 3, 6, 8);
```

This makes `a` the requested size. The new tensor is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost.

Get the extent of the various dimensions:

```
Index nl = a.nlibraries();
Index nv = a.nvitrines();
Index ns = a.nshelves();
Index nb = a.nbooks();
Index np = a.npages();
Index nr = a.nrows();
Index nc = a.ncols();
```

Which of these functions are available depends on the dimension of your tensor. For example, `nlibraries()` is only available for `Tensor7`. Note, that I took care that the first letters of the dimension names are unique, which is very convenient if you prefer short names for your variables that refer so some dimension of a tensor.

Slicing:

```
Vector x = a(0,2,1,8,3,4,joker);
// Select row 4
// on page 3
// in book 8
// on shelf 1
// in vitrine 2
// in library 0
// and copy it to the Vector x.
```

Any `Range` or `Index` expression is allowed in any of the arguments, of course.

Element access:

```
cout << a(3,4,0,0,0,0,0); // Print that element.
a(0,0,0,0,0,0,0) = 3.5;   // Assign 3.5 to this element.
```

Copying tensors:

Works exactly like copying matrices. Size of output argument is adjusted for Tensors, but must already have the correct size for TensorViews.

Assigning a scalar:

```
a = 1.0;                          // Assign 1 to all elements.
```

Mathematical operators:

You can use the operators '+=', '-=', '*/', and '/=', which operate element-vise, just as for Vector.

Maximum and minimum:

```
cout << max(a);
cout << min(a);
```

Arbitrary single-argument math functions:

The function `transform` works just like for Vector.

17.4.4 Making things appear larger than they are

Assume that you have written a function that performs some calculation for a `Tensor5`:

```
void my_function(Tensor5View x);
```

Can you call this function with a `Tensor4`? Yes, you can:

```
Tensor4 a;
Tensor5View b = a;                // The extent of the first
                                  // dimension of b will be 1.
my_function(b);                   // Call the function.
```

In general, you can always create a view that is one dimension bigger than what you have. The leading dimension then has extent 1. There is one important exception: If you interpret a `Vector` as a `Matrix`, the trailing dimension will be 1, not the leading dimension. This is necessary, because the vector has to act like a column vector, so that matrix-vector products work in the normal way. Of course you can use telescoping to blow up anything to `Tensor7`:

```
Numeric b = 3.1415;               // Just any number here.
Tensor7View bt7 =
  Tensor6View(
     Tensor5View(
        Tensor4View(
           Tensor3View(
              MatrixView(
                 VectorView(b)
              )
           )
        )
     )
  );                  // All dimensions of bt7 will be 1!
```

This kind of conversion works also implicitly. So, instead of the first example, you could have simply written:

```
my_function(a);
```

The purpose of this feature is to avoid having to make special versions of auxiliary functions for all different tensor dimensions. Use it wisely! There is very little runtime overhead for this, since the data itself is not copied.

17.4.5 Summary

This is all. In particular, we have no tensor products, since they are not needed. So, tensors are mostly used to store things, notably atmospheric field and so on. There is a set of sophisticated interpolation routines, though, which are described separately in Chapter 18.

17.5 Arrays

The template class `Array` can be used to make arrays out of anything. I do not know a good definition for 'array', but I guess anybody who has written a computer program in any programming language is familiar with the concept. Of course, it is rather similar to the concept of a Vector, just missing all the mathematical functionality like Matrix-Vector multiplication and sub-range access.

The implementation of our `Array` class is based on the STL class `std::vector`, whereas the implementation of our `Vector` class is done from scratch. So the two implementations are completely independent. Nevertheless, I tried to make `Array` behave consistently with `Vector`, as much as possible. There are a number of important differences, though, hopefully sufficiently explained in this part. A short summary of important differences:

- An Array can contain elements of any type, whereas a Vector always contains elements of type Numeric.
- No mathematical functionality for Array (no sub-ranges (nothing like VectorView); no +=, -=, *=, /=; no scalar product; no `transform` function; no `mult` function; no `transpose` function).
- On the other hand, resizing (for example adding to the end) of an Array is ok. (See the `push_back` method below.) It is still rather expensive, though, at least for large Arrays.

17.5.1 Constructing an Array

You can construct an object of an Array class like this:

```
Array<Index>  a;         // Empty Array of class Index.

Array<String> b(5);      // String Array with 5
                         // elements. Without initialization,
```

```
                          // elements contain random values.
Array<String> c(5,"x"); // The same, but fill with "x".

Array<Index>  d=a;      // Make d a copy of a;
```

There are already a lot of predefined Array classes. The naming convention for them is: `ArrayOfIndex`, `ArrayOfString`, etc.. Normally you should use these predefined classes. But if you want to define an Array of some uncommon type, you can do it with '<>', as in the above examples.

As for Vector, there is a special sub-class of Array that can be initialized explicitly. Usage:

```
MakeArray<String> a("ARTS",
                    "is",
                    "great"); // Creates an array of String
                              // with these 3 elements.
```

17.5.2 What you can do with an Array

All examples below assume that `a` is an ArrayOfString.

Resize:

```
a.resize(5);
```

This adjusts the size of `a` to 5. Resizing is more efficiently implemented than for Vector, but still expensive.

Get the number of elements:

```
cout << a.nelem();  // Just as for Vector.
```

In particular, note that the return type of this method is `Index`, just as for Vector. This is an extension compared to std::vector, which just has a method `size()` that returns the positive integer type `size_t`.

Element access:

```
cout << a[3];   // Print 4th element.
a[0] = "Hello"; // Assign string "Hello" to first element.
```

In other words, this works just like for Vector.

Copying Arrays:

This works also the same as for Vector. The size of the target must match! In this respect, I have modified the behavior with respect to the underlying std::vector, which has different copy semantics.

Assigning a scalar of the base type:

```
a = "Hello";    // Assign string "Hello" to all elements.
```

Append to the end:

```
a.push_back("Hello"); // Adds this new element at the
                      // end of a.
```

This can be an expensive operation, especially for large Arrays. Therefore, use it with care. Actually, the `push_back` method comes from the `std::vector` class that Array is based on. You can do a lot more with `std::vector`, all of which also works with `Array`. However, to explain the Standard Template Library is beyond the scope of this text. You can read about it in C++ or even dedicated STL textbooks.

17.6 Sparse matrices

The class `Sparse` implements the mathematical concept of a matrix, same as Matrix does, but the data is stored in a different manner. Sparse offers a memory saving storage when most of the matrix is filled with zeros. This means that:

- A Sparse contains floating point values of type `Numeric`.
- The values are arranged in rows and columns in the same ways as for ordinary matrices, in *row-major* order.
- A Sparse can be multiplied with a Vector, a Matrix or with another Sparse.
- There exist no views for Sparse.
- Resizing a Sparse is expensive and should be avoided.

To calculate the maximum number of non-zero elements for efficient storage, take the product of number of columns and number of rows, subtract the number of columns plus one and then divide by two, ($nnz \leq 0.5 \times (ncols \times nrows - (ncols + 1))$).

17.6.1 Constructing a Sparse

You can construct an object of class Sparse in any of these ways:

```
Sparse a;           // Create empty Sparse.
Sparse b(3,4);      // Create Sparse with 3 rows
                    // and 4 columns. When
                    // created like this it will
                    // contain only zeros, i.e.
                    // be an empty Sparse.

Sparse d=c;         // Make d a copy of c.
```

That is all. As for Matrix there exist no more fancy constructors, like an equivalent to the `MakeVector` class.

17.6.2 What you can do with a Sparse

All examples below assume that `a` is a Sparse.

Identity matrix:

```
a.make_I(10,10);
```

This sets `a` to be the identity matrix of size 10x10 (10 rows and 10 columns). Using this function is much faster than setting the diagonal elements to one by yourself. The number of rows and columns doesn't have to match each other. In the case that they don't, the rule that each row and column only will have only one position occupied by a one is applied. That is, in the case there are more rows than columns, the last rows will be empty.

Resize:

```
a.resize(5,10);
```

This makes `a` a 5x10 Sparse (5 rows, 10 columns). Note that the previous content will be completely lost. The new Sparse will be empty.

Get the number of rows, columns or non-zero elements:

```
cout << a.nrows();
cout << a.ncols();
cout << a.nnz();
```

Element access:

There are two different ways to access individual elements. One used for read only and one for read and write. The distinction is necessary since the read and write method creates elements if they don't already exist. Note that we use 0-based indexing. For reading only use:

```
cout << a.ro(3,4);  // Print that element. If it
                    // it doen't exist a zero will
                    // be printed.
cout << a(0,0);     // Short version of the above.
```

For reading and writing, such as assigning values to elements, use:

```
a.rw(0,0) = 1.5;    // Assigns the value 1.5 to the
                    // first row and first column.
cout << a.rw(0,0);  // Also returns the value of the
                    // first row and first column,
                    // if the element doesn't exist
                    // it will be created and set
                    // to zero.
```

Copying Matrices:

```
Sparse b;
b = a;
```

As in the case of Vectors, the '=' operator adjusts the size of the target automatically.

Transpose:

The function `transpose` works a bit differently for Sparse than for Vector and Matrix. This is due to the fact that we don't have any views for Sparse. Thus, `transpose` for a Sparse creates a new Sparse variable that contains the transpose of the original Sparse, whereas `transpose` for a Matrix just creates a transposed view of the original Matrix.

The target variable for the transposed Sparse has to have the right dimensions before the function is called.

```
Sparse b(a.ncols(),a.nrows());
transpose(b,a);      // Make b the transpose of a.
                     // Note the argument order!
```

Matrix multiplication:

```
// Sparse-Vector
Vector b(a.nrows()), c(a.ncols());
mult(b,a,c);         // b = a * c

// Sparse-Matrix
Matrix d(a.nrows(),5), e(a.ncols(),5);
mult(d,a,e);         // d = a * e

// Sparse-Sparse
Sparse f(a.nrows(),5), g(a.ncols(),5);
mult(f,a,g);         // f = a * g
```

The result is put in the first argument, consistent with the Matrix class. Note that for the Sparse – Matrix multiplication the output is a Matrix. **Important: As for Matrix, the matrices or vectors that you give for the three arguments must not overlap, or you will get garbage.**

Chapter 18

Interpolation

There are no general single-step interpolation functions in ARTS. Instead, there is a set of useful utility functions that can be used to achieve interpolation. Roughly, you can separate these into functions determining grid position arrays, functions determining interpolation weight tensors, and functions applying the interpolation. Doing an interpolation thus requires a chain of function calls:

1. `gridpos` (one for each interpolation dimension)
2. `interpweights`
3. `interp`

Currently implemented in ARTS is mulitlinear interpolation in up to 6 dimensions. (Is the 6D case called hexa-linear interpolation?) The necessary functions and their interaction will be explained in this chapter.

18.1 Implementation files

Variables and functions related to interpolation are defined in the files:

- `interpolation.h`
- `interpolation.cc`
- `test_interpolation.cc`

The first two files contain the declarations and implementation, the last file some usage examples.

History
020528 Created by Stefan Buehler.

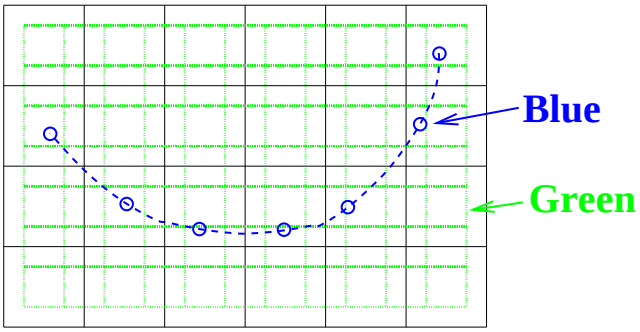

Figure 18.1: The two different types of interpolation. Green (dotted): Interpolation to a new grid, output has same dimension as input, in this case 2D. Blue (dashed): Interpolation to a sequence of points, output is always 1D.

18.2 Green and blue interpolation

There are two different types of interpolation in ARTS:

Green Interpolation: Interpolation of a gridded field to a new grid.

Blue Interpolation: Interpolation of a gridded field to a sequence of positions.

Figure 18.1 illustrates the different types for a 2D example.

The first step of an interpolation always consists in determining where your new points are, relative to the original grid. You can do this separately for each dimension. The positions have to be stored somehow, which is described in the next section.

18.3 Grid positions

A grid position specifies where an interpolation point is, relative to the original grid. It consists of three parts, an `Index` giving the original grid index below the interpolation point, a `Numeric` giving the fractional distance to the next original grid point, and a `Numeric` giving 1 minus this number. Of course, the last element is redundant. However, it is efficient to store this, since it is used many times over. We store the two numerics in a plain C array of dimension 2. (No need to use a fancy Array or Vector for this, since the dimension is fixed.) So the structure `GridPos` looks like:

```
struct GridPos  {
   Index   idx;       /*!< Original grid index below
                           interpolation point. */
   Numeric fd[2];     /*!< Fractional distance to next point
                           (0<=fd[0]<=1), fd[1] = 1-fd[0]. */
};
```

For example, `idx`=3 and `fd`=0.5 means that this interpolation point is half-way between index 3 and 4 of the original grid. Note, that ‘below’ in the first paragraph means ‘with a lower index’. If the original grid is sorted in descending order, the value at the grid point

below the interpolation point will be numerically higher than the interpolation point. In other words, grid positions and fractional distances are defined relative to the order of the original grid. Examples:

```
old grid = 2 3
new grid = 2.25
idx      = 0
fd[0]    = 0.25

old grid = 3 2
new grid = 2.25
idx      = 0
fd[0]    = 0.75
```

Note that `fd[0]` is different in the second case, because the old grid is sorted in descending order. Note also that `idx` is the same in both cases.

Grid positions for a whole new grid are stored in an `Array<GridPos>` (called `ArrayOfGridPos`).

18.4 Setting up grid position arrays

There is only one function to set up grid position arrays, namely `gridpos`:

```
void gridpos( ArrayOfGridPos& gp,
              ConstVectorView old_grid,
              ConstVectorView new_grid );
```

Some points to remember:

- As usual, the output `gp` has to have the right dimension.
- The old grid has to be strictly sorted. It can be in ascending or descending order. But there must not be any duplicate values. Furthermore, the old grid must contain at least two points.
- The new grid does not have to be sorted, but the function will be faster if it is sorted or mostly sorted. It is ok if the new grid contains only one point.
- The beauty is, that this is all it needs to do also interpolation in higher dimensions: You just have to call gridpos for all the dimensions that you want to interpolate.
- Note also, that for this step you do not need the field itself at all!

18.5 Interpolation weights

As explained in the 'Numerical Recipes' [*Press et al.*, 1997], 2D bi-linear interpolation means, that the interpolated value is a weighted average of the original field at the four corner points of the grid square in which the interpolation point is located. For simplicity,

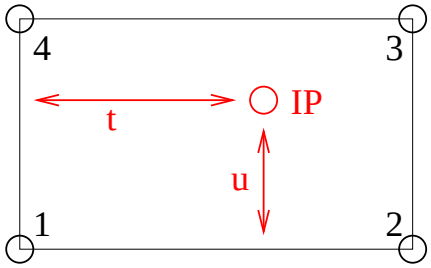

Figure 18.2: The grid square for 2D interpolation. The numbers 1...4 mark the corner points, IP is the interpolation point, t and u are the fractional distances in the two dimensions.

we label the four corner points counterclockwise, starting from the lower left point (Figure 18.2). Then the interpolated value is given by:

$$\begin{aligned} y(t,u) &= (1-t)*(1-u)*y_1 \\ &\quad + t*(1-u)*y_2 \\ &\quad + t*u*y_3 \\ &\quad + (1-t)*u*y_4 \\ &= w_1*y_1 + w_2*y_2 + w_3*y_3 + w_4*y_4 \end{aligned} \tag{18.1}$$

where t and u are the fractional distances between the corner points in the two dimensions, y_i are the field values at the corner points, and w_i are the interpolation weights.

(By the way, I have discovered that this is exactly the result that you get if you first interpolate linearly in one dimension, then in the other. I was playing around with this a bit, but it is the more efficient way to pre-calculate the w_i and do all dimensions at once.

How many interpolation weights one needs for a multilinear interpolation depends on the dimension of the interpolation: There are exactly 2^n interpolation weights for an n dimensional interpolation. These weights have have to be computed for each interpolation point (each grid point of the new grid, if we do a 'green' type interpolation. Or each point in the sequence, if we do a 'blue' type interpolation).

This means, calculating the interpolation weights is not exactly cheap, especially if one interpolates simultaneously in many dimensions. On the other hand, one can save a lot by re-using the weights. Therefore, interpolation weights in ARTS are stored in a tensor which has one more dimension than the output field. The last dimension is for the weight, so this last dimension has the extent 4 in the 2D case, 8 in the 3D case, and so on (always 2^n).

In the case of a 'blue' type interpolation, the weights are always stored in a matrix, since the output field is always 1D (a vector).

18.6 Setting up interpolation weight tensors

Interpolation weight tensors can be computed by a family of functions, which are all called `interpweights`. Which function is actually used depends on the dimension of the input and output quantities. For this step we still do not need the actual fields, just the grid positions.

18.6.1 Blue interpolation

In this case the functions are:

```
void interpweights( MatrixView itw,
                    const ArrayOfGridPos& cgp );
void interpweights( MatrixView itw,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( MatrixView itw,
                    const ArrayOfGridPos& pgp,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( MatrixView itw,
                    const ArrayOfGridPos& vgp,
                    const ArrayOfGridPos& sgp,
                    const ArrayOfGridPos& bgp,
                    const ArrayOfGridPos& pgp,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
```

In all cases, the dimension of `itw` must be consistent with the given grid position arrays and the dimension of the interpolation (last dimension 2^n). Because the grid position arrays are interpreted as defining a sequence of positions they must all have the same length.

18.6.2 Green interpolation

In this case the functions are:

```
void interpweights( Tensor3View itw,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( Tensor4View itw,
                    const ArrayOfGridPos& pgp,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( Tensor5View itw,
                    const ArrayOfGridPos& bgp,
                    const ArrayOfGridPos& pgp,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( Tensor6View itw,
                    const ArrayOfGridPos& sgp,
                    const ArrayOfGridPos& bgp,
                    const ArrayOfGridPos& pgp,
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
void interpweights( Tensor7View itw,
                    const ArrayOfGridPos& vgp,
                    const ArrayOfGridPos& sgp,
                    const ArrayOfGridPos& bgp,
                    const ArrayOfGridPos& pgp,
```

```
                    const ArrayOfGridPos& rgp,
                    const ArrayOfGridPos& cgp );
```

In this case the grid position arrays are interpreted as defining the grids for the interpolated field, therefore they can have different lengths. Of course, `itw` must be consistent with the length of all the grid position arrays, and with the dimension of the interpolation (last dimension 2^n).

18.7 The actual interpolation

For this final step we need the grid positions, the interpolation weights, and the actual fields. For each interpolated value, the weights are applied to the appropriate original field values and the sum is taken (see Equation 18.1). The `interp` family of functions performs this step.

18.7.1 Blue interpolation

```
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstVectorView       a,
             const ArrayOfGridPos& cgp);
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstMatrixView       a,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstTensor3View      a,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstTensor4View      a,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstTensor5View      a,
             const ArrayOfGridPos& sgp,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( VectorView            ia,
             ConstMatrixView       itw,
             ConstTensor6View      a,
```

```
             const ArrayOfGridPos& vgp,
             const ArrayOfGridPos& sgp,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
```

18.7.2 Green interpolation

```
void interp( MatrixView            ia,
             ConstTensor3View      itw,
             ConstMatrixView       a,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( Tensor3View           ia,
             ConstTensor4View      itw,
             ConstTensor3View      a,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( Tensor4View           ia,
             ConstTensor5View      itw,
             ConstTensor4View      a,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( Tensor5View           ia,
             ConstTensor6View      itw,
             ConstTensor5View      a,
             const ArrayOfGridPos& sgp,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
void interp( Tensor6View           ia,
             ConstTensor7View      itw,
             ConstTensor6View      a,
             const ArrayOfGridPos& vgp,
             const ArrayOfGridPos& sgp,
             const ArrayOfGridPos& bgp,
             const ArrayOfGridPos& pgp,
             const ArrayOfGridPos& rgp,
             const ArrayOfGridPos& cgp);
```

18.8 Examples

18.8.1 A simple example

This example is contained in file `test_interpolation.cc`.

```
void test05()
{
  cout << "Very simple interpolation case\n";

  Vector og(1,5,+1);            // 1, 2, 3, 4, 5
  Vector ng(2,5,0.25);          // 2.0, 2,25, 2.5, 2.75, 3.0

  cout << "Original grid:\n" << og << "\n";
  cout << "New grid:\n" << ng << "\n";

  // To store the grid positions:
  ArrayOfGridPos gp(ng.nelem());

  gridpos(gp,og,ng);
  cout << "Grid positions:\n" << gp;

  // To store interpolation weights:
  Matrix itw(gp.nelem(),2);
  interpweights(itw,gp);

  cout << "Interpolation weights:\n" << itw << "\n";

  // Original field:
  Vector of(og.nelem(),0);
  of[2] = 10;                   // 0, 0, 10, 0, 0

  cout << "Original field:\n" << of << "\n";

  // Interpolated field:
  Vector nf(ng.nelem());

  interp(nf, itw, of, gp);

  cout << "New field:\n" << nf << "\n";
}
```

Ok, maybe you think this is not so simple, but a large part of the code is either setting up the example grids and fields, or output. And here is how the output looks like:

```
Very simple interpolation case
Original grid:
  1   2   3   4   5
New grid:
  2 2.25 2.5 2.75   3
Grid positions:
   1 0    1
   1 0.25 0.75
   1 0.5  0.5
   1 0.75 0.25
   1 1    0
Interpolation weights:
  1   0
```

```
0.75 0.25
0.5 0.5
0.25 0.75
  0   1
Original field:
  0   0  10   0   0
New field:
  0 2.5   5 7.5  10
```

18.8.2 A more elaborate example

What if you want to interpolate only some dimensions of a tensor, while retaining others? — You have to make a loop yourself, but it is very easy. Below is an explicit example for a more complicated interpolation case. (Green type interpolation of all pages of a Tensor3.) This example is also contained in file `test_interpolation.cc`.

```
void test04()
{
  cout << "Green type interpolation of all "
       << "pages of a Tensor3\n";

  // The original Tensor is called a, the new one n.

  // 10 pages, 20 rows, 30 columns, all grids are: 1,2,3
  Vector  a_pgrid(1,3,1), a_rgrid(1,3,1), a_cgrid(1,3,1);
  Tensor3 a( a_pgrid.nelem(),
             a_rgrid.nelem(),
             a_cgrid.nelem() );
  a = 0;
  // Put some simple numbers in the middle of each page:
  a(0,1,1) = 10;
  a(1,1,1) = 20;
  a(2,1,1) = 30;

  // New row and column grids:
  // 1, 1.5, 2, 2.5, 3
  Vector  n_rgrid(1,5,.5), n_cgrid(1,5,.5);
  Tensor3 n( a_pgrid.nelem(),
             n_rgrid.nelem(),
             n_cgrid.nelem() );

  // So, n has the same number of pages as a,
  // but more rows and columns.

  // Get the grid position arrays:
  ArrayOfGridPos n_rgp(n_rgrid.nelem()); // For rows.
  ArrayOfGridPos n_cgp(n_cgrid.nelem()); // For columns.

  gridpos( n_rgp, a_rgrid, n_rgrid );
  gridpos( n_cgp, a_cgrid, n_cgrid );
```

```
  // Get the interpolation weights:
  Tensor3 itw( n_rgrid.nelem(), n_cgrid.nelem(), 4 );
  interpweights( itw, n_rgp, n_cgp );

  // Do a "green" interpolation for all pages of a:

  for ( Index i=0; i<a.npages(); ++i )
    {
      // Select the current page of both a and n:
      ConstMatrixView ap = a( i,
                              Range(joker), Range(joker) );
      MatrixView      np = n( i,
                              Range(joker), Range(joker) );

      // Do the interpolation:
      interp( np, itw, ap, n_rgp, n_cgp );

      // Note that this is efficient, because interpolation
      // weights and grid positions are re-used.
    }

  cout << "Original field:\n";
  for ( Index i=0; i<a.npages(); ++i )
      cout << "page " << i << ":\n"
           << a(i,Range(joker),Range(joker)) << "\n";

  cout << "Interpolated field:\n";
  for ( Index i=0; i<n.npages(); ++i )
      cout << "page " << i << ":\n"
           << n(i,Range(joker),Range(joker)) << "\n";
}
```

The output is:

```
Green type interpolation of all pages of a Tensor3
Original field:
page 0:
  0   0   0
  0  10   0
  0   0   0
page 1:
  0   0   0
  0  20   0
  0   0   0
page 2:
  0   0   0
  0  30   0
  0   0   0
Interpolated field:
page 0:
  0   0   0   0   0
  0 2.5   5 2.5   0
```

```
  0   5  10   5   0
  0 2.5   5 2.5   0
  0   0   0   0   0
page 1:
  0   0   0   0   0
  0   5  10   5   0
  0  10  20  10   0
  0   5  10   5   0
  0   0   0   0   0
page 2:
  0   0   0   0   0
  0 7.5  15 7.5   0
  0  15  30  15   0
  0 7.5  15 7.5   0
  0   0   0   0   0
```

18.9 Higher order interpolation

Everything that was written so far in this chapter referred to linear interpolation, which uses 2 neighboring data points in the 1D case. But ARTS has also a framework for higher order polynomial interpolation. It is defined in the two files

- `interpolation_poly.h`
- `interpolation_poly.cc`

We define interpolation order O as the order of the polynomial that is used. Linear interpolation, the ARTS standard case, corresponds to $O = 1$. $O = 2$ is quadratic interpolation, $O = 3$ cubic interpolation. The number of interpolation points (and weights) for a 1D interpolation is $O + 1$ for each point in the new grid. So, linear interpolation uses 2 points, quadratic 3, and cubic 4.

Note, that if you use even interpolation orders, you will have an unequal number of interpolation points 'to the left' and 'to the right' of your new point. This is an argument for preferring $O = 3$ as the basic higher order polynomial interpolation, instead of $O = 2$.

Overall, higher order interpolation works rather similarly to the linear case. The main difference is that grid positions for higher order interpolation are stored in an object of type `GridPosPoly`, instead of `GridPos`. A `GridPosPoly` object contains grid indices and interpolation weights for all interpolation points. For each point in the new grid, there are $O + 1$ indices and $O + 1$ weights.

The reason why we store all interpolation point indices, and not only the index of the first point, is to allow correct handling of circular interpolation, for example in scattering phase function ϕ angle. If the angle goes from 0 to 360°, then points just below 360 should be used in interpolations to points just above 0, so the indices to use are not contiguous in memory. Functions to handle this are not yet implemented, but this should be a relatively simple matter.

In contrast to `GridPos`, `GridPosPoly` stores weights `w` rather than fractional distances `fd`. For the linear case:

```
  w[0] = fd[1]
  w[1] = fd[0]
```

So the two concepts are almost the same. Because the w are associated with each interpolation point, they work also for higher interpolation order, whereas the concept of fractional distance does not.

The weights are calculated according to section 3.1, eq. 3.1.1 of [*Press et al.*, 1997]. These are for the 1D case. For 2D and higher dimensional cases, the weights of the individual dimensions have to be multiplied, just as in the linear interpolation case.

Instead of `gridpos`, you have to use the function `gridpos_poly` for higher order interpolation. It works exactly like `gridpos`, but has an additional argument that gives the interpolation order O.

After setting up the `GridPosPoly` object with `gridpos_poly`, you have to call `interpweights` and `interp`, exactly as in the linear case. (The actual functions used are not the same, since the name is overloaded. The `interpweights` and `interp` functions for use with `GridPosPoly` are implemented in `interpolation_poly.cc`.) So, a complete interpolation chain involves:

```
gridpos_poly
interpweights
interp
```

For $O = 1$ the result of the interpolation chain will be the same as for the linear interpolation routines. Below is a simple complete example, taken from the file `test_interpolation.cc` in the arts source directory:

```
void test08()
{
  cout << "Very simple interpolation case for the "
       << "new higher order polynomials.\n";

  Vector og(1,5,+1);              // 1, 2, 3, 4, 5
  Vector ng(2,5,0.25);            // 2.0, 2,25, 2.5, 2.75, 3.0

  cout << "Original grid:\n" << og << "\n";
  cout << "New grid:\n" << ng << "\n";

  // To store the grid positions:
  ArrayOfGridPosPoly gp(ng.nelem());

  Index order=2;                  // Interpolation order.

  gridpos_poly(gp,og,ng,order);
  cout << "Grid positions:\n" << gp;

  // To store interpolation weights:
  Matrix itw(gp.nelem(),order+1);
```

```
  interpweights(itw,gp);

  cout << "Interpolation weights:\n" << itw << "\n";

  // Original field:
  Vector of(og.nelem(),0);
  of[2] = 10;                    // 0, 0, 10, 0, 0

  cout << "Original field:\n" << of << "\n";

  // Interpolated field:
  Vector nf(ng.nelem());

  interp(nf, itw, of, gp);

  cout << "New field (order=" << order << "):\n" << nf << "\n";

  cout << "All orders systematically:\n";
  for (order=1; order<5; ++order)
    {
      gridpos_poly(gp,og,ng,order);
      itw.resize(gp.nelem(),order+1);
      interpweights(itw,gp);
      interp(nf, itw, of, gp);

      cout << "order " << order << ": ";
      for (Index i=0; i<nf.nelem(); ++i)
        cout << setw(8) << nf[i] << " ";
      cout << "\n";
    }
}
```

18.10 Summary

Now you probably understand better what was written at the very beginning of this chapter, namely that doing an interpolation always requires the chain of function calls:

1. `gridpos` or `gridpos_poly` (one for each interpolation dimension)
2. `interpweights`
3. `interp`

If you are interested in how the functions really work, look in file `interpolation.cc` or `interpolation_poly.cc`. The documentation there is quite detailed. When you are using interpolation, you should always give some thought to whether you can re-use grid positions or even interpolation weights. This can really save you a lot of computation

time. For example, if you want to interpolate several fields — which are all on the same grids — to some position, you only have to compute the weights once.

Chapter 19

Integration functions

A radiative transfer model which takes into account the effect of scattering involves integration of certain quantities over the angles of observation. For example from Section 13.1.2 it is clear that computing scattering cross-section and scattering integral term requires integration over zenith and azimuth directions. There are a wide range of methods that can be used for numerical integration. They can be used depending on various factors starting from how accurate the result should be to the behaviour of the function. The one which is implemented in ARTS is the trapezoidal integration method.

19.1 Implementation files

The integration functions can be found in the files:

- `math_funcs.h`
- `math_funcs.cc`

The implementation function `AngIntegrate_trapezoid`is discussed in the second file.

19.2 Trapezoidal Integration

Trapezoidal Integration method comes under the Newton-Cotes formulas where integration of a function is approximated by the area under the curve described by the function. Trapezoidal integration assumes that the area under the curve is trapezoid.

Trapezoidal rule :

$$\int_{x_1}^{x_2} f(x)dx = \frac{1}{2}h(f_1 + f_2) + O(h^3 f'') \tag{19.1}$$

History	
220802	Created and written by Sreerekha T.R.
220103	Included mathematical description for implemented integration method(CE).

This is a two-point formula (x_1 and x_2). It is exact for polynomials upto and including degree 1, i.e., f(x) = x. $O(h^3 f^{''})$ signifies how far is the true answer from the estimate.

If we use eq. 19.1 $N-1$ times, to do the integration in the intervals (x_1, x_2), (x_2, x_3), ..., (x_{N-1}, x_N), and then add the results, we obtain extended formula for the integral from x_1 to x_N.

Extended Trapezoidal rule :

$$\int_{x_1}^{x_N} f(x)dx = \frac{1}{2}h\left[f_1 + 2(f_2 + f_3 + ... + f_{N-1}) + f_N\right] + O\left[\frac{(b-a)^3 f^{''}}{N^2}\right] \quad (19.2)$$

The last term tells how much the error will be decreased by taking more number of steps.

19.3 Solid Angle Integration

In our scattering problem, we are often encountered with a double integration of functions over zenith and azimuth angles (see Chapter 13). One way to achieve double integration is to use repeated one-dimensional trapezoidal integration. This is effective of course only if the boundary is simple and the function is very smooth. If the function is strongly peaked and if know where it occurs, integral should be broken into smaller regions so that the integrand is smooth in each. Another thing is to take into account the symmetry of the function as well as the boundary. For example in our case, if the radiation is symmetric about the azimuth, the integration in that direction returns constant value of 2π and we need to do only integration over zenith directions.

The general form of a solid angle integration is

$$S = \int_{4\pi} f(\omega)\mathrm{d}\omega \quad (19.3)$$

In spherical coordinates we can write:

$$S = \int_0^{\pi}\int_0^{2\pi} f(\theta,\phi)\sin\theta \quad \mathrm{d}\theta\mathrm{d}\phi \quad (19.4)$$

A double integration can be splitted into two single integrations:

$$S = \int_0^{\pi}\left(\int_0^{2\pi} f(\theta,\phi)\sin\theta\mathrm{d}\phi\right)\mathrm{d}\theta \quad (19.5)$$

$$= \int_0^{\pi} g(\theta)\mathrm{d}\theta \quad (19.6)$$

If we have to integrate a vector, we can apply this method componentwise.

To solve the integral numerically we discretize θ and ϕ and obtain two angular grids ($[\theta_0, \theta_1, \cdots, \theta_n]$ and $[\phi_0, \phi_1, \cdots, \phi_m]$). Then we can first calculate $g(\theta_j)$ for all θ_j unsing the trapezoidal method.

$$g(\theta_j) = \sum_{i=1}^{m} \sin\theta_j \frac{f(\theta_j,\phi_i) + f(\theta_j,\phi_{i+1})}{2} \cdot (\phi_{i+1} - \phi_i) \quad (19.7)$$

The final step is to sum up all $g(\theta_j)$, again applying the trapezoidal method.

$$S = \sum_{j=1}^{n} \frac{g(\theta_j) + g(\theta_{j+1})}{2} \cdot (\theta_{j+1} - \theta_j) \tag{19.8}$$

If the radiation is symmetric about the azimuth we just calculate:

$$S_{sym} = 2\pi \int_0^{\pi} f(\theta) \sin(\theta) \mathrm{d}\theta \tag{19.9}$$

Unsing the trapezoidal method this can be written as:

$$S_{sym} = 2\pi \sum_{j=1}^{n} \frac{h(\theta_j) + h(\theta_{j+1})}{2} \cdot (\theta_{j+1} - \theta_j) \tag{19.10}$$

where $h(\theta) = \sin\theta \cdot f(\theta)$.

The function `AngIntegrate_trapezoid` takes as input the integrand and the angles over which the integration has to be done. For example in this case it can be the zenith and azimuth angle grid.

```
Numeric AngIntegrate_trapezoid(MatrixView Integrand,
                               ConstVectorView za_grid,
                               ConstVectorView aa_grid)
```

The integrand has the same number of rows as zenith angle grid and columns as azimuth angle grid. The inner loop does trapezoidal integration of the integrand over all azimuth angles and the result is stored in a Vector res1[i]. Note that the integrand at every point has to be multiplied with `sin (za_grid[i] * DEG2RAD)` since we are integrating over solid angles. The outer loop does an integration of res1[i] over all zentih angles. The result of this is returned back to the calling function.

Chapter 20

Linear algebra functions

Solving the vector radiative transfer equation requires the computation of linear equation systems and the matrix exponential. This section describes the functions which are implemented in ARTS and it gives instructions how these functions can be used, also for other purposes than the radiative transfer calculations.

20.1 Implementation files

All the functions described below can be found in the files:

- `lin_alg.h`
- `lin_alg.cc`

The template class `Array` and the classes `Matrix` and `Vector` are used, therefore the linear algebra functions require the files:

- `matpackI.h`
- `make_vector.h`
- `array.h`
- `matpackI.cc`
- `make_vector.cc`
- `array.cc`

Furthermore logical functions contained in

- `logic.h`
- `logic.cc`

are used to check the dimensions of input matrices for various functions.

History
020502 Created and written by Claudia Emde.

20.2 Linear Equation Systems

For solving a set of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{20.1}$$

the LU decomposition method is implemented.A slightly modified version of the algorithm described in [*Press et al.* [1997]] is used here. An alternative method is the Gauss-Jordan elimination, but this method is three times slower than the LU decomposition method [*Press et al.* [1997], p.36]. The LU decomposition method reqires two functions, `ludcmp` and `lubacksub`, which will be decribed below.

The following example for a three dimensional equation sytem demonstrates how to solve a linear equation sytem of the type (20.1):

- Create matrix A, vector b:
```
A = Matrix(3,3);
A(1,1) = 4;
A(2,1) = 3;
...
b = Vector(3);
b(1) = 7;
...
```
- Initialize solution vector x and two other variables needed for storing intermediate results:
```
x = Vector(3);
LU = Matrix(3,3);
indx = ArrayOfIndex(3);
```
- Call LU decomposition function (see Section 20.2.1):
```
ludcmp(LU, indx, A);
```
- Call LU backsubstitution function (see Section 20.2.2):
```
lubacksub(x, LU, b, indx);
```
- Print the solution vector:
```
cout << x;
```

20.2.1 LU Decomposition

A LU decomposition is a procedure for decomposing a square matrix $\mathbf{A}$ with dimension n into a product of a lower triangular matrix $\mathbf{L}$ (has elements only on the diagonal elements and below) and an upper triangular matrix $\mathbf{U}$ (has elements only on the diagonal and above):

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{20.2}$$

For a 3 x 3 matrix equation 20.2 would look like this:

$$\begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \cdot \begin{pmatrix} u11 & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

The decomposition can be used to rewrite the linear set of equations (20.1) in the following way:

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{20.3}$$

First

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{20.4}$$

is solved for the vector $\mathbf{y}$ which can be done by forward substitution (see section 20.2.2). Then

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{20.5}$$

is solved again by backsubstitution. The advantage in breaking up one linear set into two successive ones is that the solution of a triangular set of equations is quite trivial.

The function `ludcmp` requires a square matrix of arbitrary dimension n as input and performs the LU decomposition. It returns one matrix which contains both matrices, $\mathbf{L}$ and $\mathbf{U}$. For the lower triangular matrix $\mathbf{L}$ the diagonal elements are chosen to be 1, then the other elements of $\mathbf{L}$ and $\mathbf{U}$ are determined. This is possible, as the LU decomposition is an under determined equation sytem with n^2 equations for $n^2 + n$ unknowns. The output matrix does not include the diagonal of $\mathbf{L}$, in the three-dimensional case it has the following elements:

$$\begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{pmatrix}$$

This special arrangement of the LU decomposition is named *Crout's algorithm* and a matrix arranged in this form is named *Crout matrix* in this context.

Another output variable of the function `ludcmp` is an index vector which contains information about pivoting which is absolutely essential for the stability of Crout's algorithm. Here partial pivoting, i.e. interchange of rows is implemented. That means that not $\mathbf{A}$ is decomposed into LU-form but a rowwise permutation of $\mathbf{A}$. If the index vector contains for example the elements $(2, 1, 0)$ the first and the last row of a three dimensional matrix would be exchanged.

20.2.2 Forward- and Backsubstitution

An equation system of the form

$$\begin{pmatrix} a11 & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

can be solved very easy. The last element, here x_3, is already isolated, namely

$$x_3 = b_3/a_{33} \tag{20.6}$$

As x_3 is known x_2 can be calculated using the second row of the eqautions. Then, finally, x_1 can be calculated as well using the first row. This procedure is called backsubtitution. The

same method applied for an equation system including a lower triangular matrix is named forward substitution.

The function `lubacksub` does forward and backward substitution to solve the equation system described in 20.2.1. As input it requires the output variables of `ludcmp` which are the *Crout matrix* and the index vector. Output of the function is the solution vector $\mathbf{x}$ to the equation system.

20.2.3 More Applications of the LU Decomposition

- Inverse of a matrix:
  To compute $(\mathbf{K})^{-1} \cdot \mathbf{b}$, which is a part of the solution to the vector radiative transfer equation (13.10) the LU decomposition method can be used. The following equations show, that the problem is equivalent to solving a linear equation system of the type 20.1.

$$\mathbf{K}^{-1} \cdot \mathbf{b} = \mathbf{x} \tag{20.7}$$

$$\Leftrightarrow \quad \mathbf{K} \cdot \mathbf{x} = \mathbf{b} \tag{20.8}$$

- To solve the equation system

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{20.9}$$

  where **A**, **B** and **X** are matrices of dimension n, the LU decomposition functions can be applied as well. Assume that **A** and **B** are known and you want to solve for **X**. First you should do a LU decomposition of **A** and then backsubstitute with the columns of B and you get the columns of **X** as solution vectors.

20.3 Matrix Exponential Function

A very important function for solving differential equations is the matrix exponential:

$$e^{\mathbf{A}s} = \sum_{k=0}^{\infty} \frac{(\mathbf{A}s)^k}{k!} \tag{20.10}$$

In principle it could be computed using the Taylor power series but this method is not efficient. MOLER and VAN LOAN have shown for the simple example [*Moler and Loan* [1979]]

$$\mathbf{A} = \begin{pmatrix} -49 & 24 \\ -64 & 31 \end{pmatrix}$$

that convergence is obtained not until 59 terms. And if a relative accuracy of only 10^{-5} is taken, the method even leads to a wrong result due to rounding errors.

20.3.1 Padé Approximation

One of the better algorithms for computing the matrix exponential is the Padé approximation which is also shortly described in [*Moler and Loan* [1979]] and outlined in the book "Matrix Computations" by *Golub and Loan* [1991]. The method uses perturbation theorie as well as the so called Padé functions. It is possible to derive an algorithm which calculates

$$\mathbf{F} = e^{\mathbf{A}+\mathbf{E}} \tag{20.11}$$

where

$$\|\mathbf{E}\|_\infty \leq \delta \|\mathbf{A}\|. \tag{20.12}$$

The accuracy of the computation given by δ can be chosen. The parameter q has to be the smallest non-negative integer such that $\epsilon(q,q) \leq \delta$ where

$$\epsilon(p,q) = 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}. \tag{20.13}$$

The following table shows values of epsilon for different values of q.

q	ϵ(q,q)
1	0.1667
2	$6.9444 \cdot 10^{-4}$
3	$1.2401 \cdot 10^{-6}$
4	$1.2302 \cdot 10^{-9}$
5	$7.7667 \cdot 10^{-13}$
6	$3.3945 \cdot 10^{-16}$

The algorithm is implemented in the function `matrix_exp`. Input to this function is the matrix $\mathbf{A}$ and the parameter q. As output it gives the matrix $\mathbf{F}$ which is defined above.
The following example shows how to use the `matrix_exp` function:

- Initialize **A** and assign values:
```
Matrix A(3,3);
A(1,1) = 45;
A(1,2) = 3;
...
```
- Initialize **F**:
```
Matrix F(3,3);
```
- Give a paramater for the accuracy:
```
Index q=6;
```
- Call the matrix exponential function:
```
matrix_exp(F,A,q);
```
- Print the result:
```
cout << "exp(A) = " << F;
```

Part V

Theoretical background

Chapter 21

Basic radiative transfer theory

When dealing with atmospheric radiation a division can be made between two different wavelength ranges where the limit is found around 5 μm, i.e. one range consists of the near IR, visible and UV regions while the second range covers thermal and far IR and microwaves. The first reason to this division is the principal sources to the radiation in the two ranges, for wavelengths shorter than 5 μm the solar radiation is dominating while at longer wavelengths the thermal emission from the surface and the atmosphere is more important. A second reason is the importance of scattering but here it is impossible to give a fixed limit. Clouds are important scattering objects for most frequencies but at cloud free conditions scattering can in many cases be neglected for wavelengths > 5 μm. If the atmosphere can be assumed to be in local thermodynamic equilibrium the radiative transfer can be simplified considerably, and this is a valid assumption for the IR region and microwaves but not for e.g. UV frequencies.

The radiative transfer in the atmosphere must be adequately described in many situations, as when estimating rates of photochemical reactions, calculating radiative forcing in the atmosphere or evaluating an remote sensing observation. It is not totally straightforward to quantify the radiative transfer with good accuracy because the calculations can be very computationally demanding and many of the parameters needed are hard to determine. For example, situations when a great number of transitions or multiple scattering must be considered will cause long calculations while as a rule scattering is problematic to model because the shape and size distribution of the scattering particles are highly variable quantities.

This chapter introduces the theoretical background which is essential to develop a radiative transfer model including scattering. The theory is based on concepts of electrodynamics, starting from the Maxwell equations. An elementary book for electrodynamics is written by *Jackson* [1998]. For optics and scattering of radiation by small particles the reader may refer for instance to *van de Hulst* [1957] and *Bohren and Huffman* [1998]. The notation used in this chapter is mostly adapted from the book "Scattering, Absorption, and Emission of Light by Small Particles" by *Mishchenko et al.* [2002]. Several lengthy deriva-

History	
050224	Copied chapter 1 from Claudia Emde's phd-thesis.
030305	Copied from a compendium written by Patrick Eriksson.

tions of formulas, which are not shown in detail here, can also be found in this book. The purpose of this chapter is to provide definitions and give ideas, how these definitions can be derived using principles of electromagnetic theory. For the derivation of the radiative transfer equation an outline of the traditional phenomenological approach is given.

21.1 Basic definitions

From the Maxwell equations one can derive the formula for the electromagnetic field vector $\mathbf{E}$ of a plane electromagnetic wave propagating in a homogeneous medium without sources:

$$\mathbf{E}(\mathbf{r},t) = \mathbf{E}_0 \exp\left(-\frac{\omega}{c} m_{\mathrm{I}} \hat{\mathbf{n}} \cdot \mathbf{r}\right) \exp\left(i\frac{\omega}{c} m_{\mathrm{R}} \hat{\mathbf{n}} \cdot \mathbf{r} - i\omega t\right), \tag{21.1}$$

where $\mathbf{E}_0$ is the amplitude of the electromagnetic wave in vacuum, c is the speed of light in vacuum, ω is the angular frequency, $\mathbf{r}$ is the position vector and $\hat{\mathbf{n}}$ is a real unit vector in the direction of propagation. The complex refractive index m is

$$m = m_{\mathrm{R}} + i m_{\mathrm{I}} = c\sqrt{\epsilon\mu}, \tag{21.2}$$

where m_{R} is the non-negative real part and m_{I} is the non-negative imaginary part. Furthermore μ is the permeability of the medium and ϵ the permittivity. For a vacuum, $m = m_{\mathrm{R}} = 1$. The imaginary part of the refractive index, if it is non-zero, determines the decay of the amplitude of the wave as it propagates through the medium, which is thus absorbing. The real part determines the phase velocity $v = c/m_{\mathrm{R}}$. The time-averaged Poynting vector $\mathbf{P}(\mathbf{r})$, which describes the flow of electromagnetic energy, is defined as

$$\mathbf{P}(\mathbf{r}) = \frac{1}{2}\mathrm{Re}(\langle \mathbf{E}(\mathbf{r})\rangle \times \langle \mathbf{H}^*(\mathbf{r})\rangle), \tag{21.3}$$

where $\mathbf{H}$ is the magnetic field vector and the $*$ denotes the complex conjugate. The Poynting vector for a homogeneous wave is given by

$$\langle \mathbf{P}(\mathbf{r})\rangle = \frac{1}{2}\mathrm{Re}\left(\sqrt{\frac{\epsilon}{\mu}}\right) |\mathbf{E}_0|^2 \exp\left(-2\frac{\omega}{c} m_{\mathrm{I}} \hat{\mathbf{n}} \cdot \mathbf{r}\right) \hat{\mathbf{n}}. \tag{21.4}$$

Equation (21.4) shows that the energy flows in the direction of propagation and its absolute value $I(\mathbf{r}) = |\langle \mathbf{P}(\mathbf{r})\rangle|$, which is usually called intensity (or irradiance), is exponentially attenuated. Rewriting Equation (21.4) gives

$$I(\mathbf{r}) = I_0 \exp(-\alpha^p \hat{\mathbf{n}} \cdot \mathbf{r}), \tag{21.5}$$

where I_0 is the intensity for $\mathbf{r} = \mathbf{0}$. The absorption coefficient α^p is

$$\alpha^p = 2\frac{\omega}{c} m_{\mathrm{I}} = \frac{4\pi m_{\mathrm{I}}}{\lambda} = \frac{4\pi m_{\mathrm{I}} \nu}{c}, \tag{21.6}$$

where λ is the free-space wavelength and ν the frequency. Intensity has the dimension of monochromatic flux [energy/(area $\times$ time)].

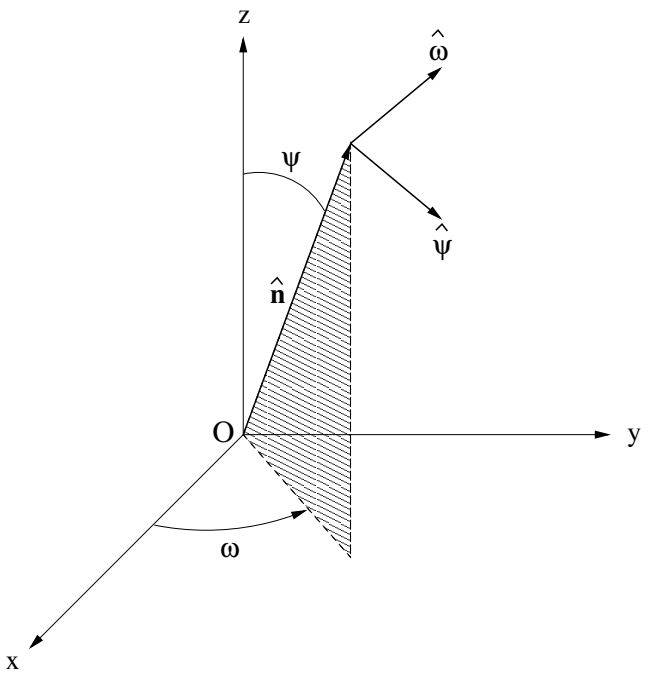

Figure 21.1: Coordinate system to describe the direction of propagation and the polarization state of a plane electromagnetic wave (adapted from Mishchenko).

21.2 The Stokes parameters

Sensors usually do not measure directly the electric and the magnetic fields associated with a beam of radiation. They measure quantities that are time averages of real-valued linear combinations of products of field vector components and have the dimension of intensity. Examples of such observable quantities are the Stokes parameters. Figure 21.1 shows the coordinate system used to describe the direction of propagation $\hat{\mathbf{n}}$ and the polarization state of a plane electromagnetic wave. The unit vector $\hat{\mathbf{n}}$ can equivalently be described by a couplet (ψ, ω), where $\psi \in [0, \pi]$ is the polar (zenith) angle and $\omega \in [0, 2\pi)$ is the azimuth angle. The electric field at the observation point is given by $\mathbf{E} = \mathbf{E}_\psi + \mathbf{E}_\omega$, where $\mathbf{E}_\psi$ and $\mathbf{E}_\omega$ are the ψ- and ω-components of the electric field vector. $\mathbf{E}_\psi$ lies in the meridional plane, which is the plane through $\hat{\mathbf{n}}$ and the z-axis, and $\mathbf{E}_\omega$ is perpendicular to this plane. The Stokes parameters are defined as linear combinations of products of the amplitudes $\mathbf{E}_\psi$ and $\mathbf{E}_\omega$ which form the 4×1 column vector $\mathbf{I}$, which is known as the Stokes vector. Since the Stokes parameters are real-valued and have the dimension of intensity, they can be measured directly with suitable instruments. The Stokes parameters are a complete set of quantities needed to characterize a plane electromagnetic wave. They carry information of the complex amplitudes and the phase difference. The first Stokes parameter I is the intensity and the other components Q, U and V describe the polarization state of the wave. For a detailed definition of the Stokes parameters and how they can be measured refer to Section 22.

21.3 Scattering, absorption and thermal emission by a single particle

A parallel monochromatic beam of electromagnetic radiation propagates in vacuum without any change in its intensity or polarization state. A small particle, which is interposed into the beam, can cause several effects:

Absorption: The particle converts some of the energy contained in the beam into other forms of energy.

Elastic scattering: Part of the incident energy is extracted from the beam and scattered into all spatial directions at the frequency of the incident beam. Scattering can change the polarization state of the radiation.

Extinction: The energy of the incident beam is reduced by an amount equal to the sum of absorption and scattering.

Dichroism: The change of the polarization state of the beam as it passes a particle.

Thermal emission: If the temperature of the particle is non-zero, the particle emits radiation in all directions over a large frequency range.

The beam is an oscillating plane magnetic wave, whereas the particle can be described as an aggregation of a large number of discrete elementary electric charges. The incident wave excites the charges to oscillate with the same frequency and thereby radiate secondary electromagnetic waves. The superposition of these waves gives the total elastically scattered field.

One can also describe the particle as an object with a refractive index different from that of the surrounding medium. The presence of such an object changes the electromagnetic field that would otherwise exist in an unbounded homogeneous space. The difference of the total field in the presence of the object can be thought of as the field *scattered* by the object. The angular distribution and the polarization of the scattered field depend on the characteristics of the incident field as well as on the properties of the object as its size relative to the wavelength and its shape, composition and orientation.

21.3.1 Definition of the amplitude matrix

For the derivation of a relation between the incident and the scattered electric field we consider a finite scattering object in the form of a single body or a fixed aggregate embedded in an infinite homogeneous, isotropic and non-absorbing medium. We assume that the individual bodies forming the scattering object are sufficiently large that they can be characterized by optical constants appropriate to bulk matter, not to optical constants appropriate for single atoms or molecules. Solving the Maxwell equations for the internal volume, which is the interior of the scattering object, and the external volume one can derive a formula, which expresses the total electric field everywhere in space in terms of the incident field and the field inside the scattering object. Applying the far field approximation gives a relation between incident and scattered field, which is that of a spherical wave. The amplitude matrix

$\mathbf{S}(\hat{\mathbf{n}}^{\mathrm{sca}}, \hat{\mathbf{n}}^{\mathrm{inc}})$ includes this relation:

$$\begin{bmatrix} E_{\psi}^{\mathrm{sca}}(r\hat{\mathbf{n}}^{\mathrm{sca}}) \\ E_{\omega}^{\mathrm{sca}}(r\hat{\mathbf{n}}^{\mathrm{sca}}) \end{bmatrix} = \frac{e^{ikr}}{r}\mathbf{S}(\hat{\mathbf{n}}^{\mathrm{sca}}, \hat{\mathbf{n}}^{\mathrm{inc}}) \begin{bmatrix} E_{0\psi}^{\mathrm{inc}} \\ E_{0\omega}^{\mathrm{inc}} \end{bmatrix}. \tag{21.7}$$

The amplitude matrix depends on the directions of incident $\hat{\mathbf{n}}^{\mathrm{inc}}$ and scattering $\hat{\mathbf{n}}^{\mathrm{sca}}$ as well as on size, morphology, composition, and orientation of the scattering object with respect to the coordinate system. The distance between the origin and the observation point is denoted by r and the wave number of the external volume is denoted by k.

The amplitude matrix provides a complete description of the scattering pattern in the far field zone. The amplitude matrix explicitly depends on ω^{inc} and ω^{sca} even when ψ^{inc} and/or ψ^{sca} equal 0 or π.

21.3.2 Phase matrix

The phase matrix $\mathbf{Z}$ describes the transformation of the Stokes vector of the incident wave into that of the scattered wave for scattering directions away from the incidence direction ($\hat{\mathbf{n}}^{\mathrm{sca}} \neq \hat{\mathbf{n}}^{\mathrm{inc}}$),

$$\mathbf{I}^{\mathrm{sca}}(r\hat{\mathbf{n}}^{\mathrm{sca}}) = \frac{1}{r^2}\mathbf{Z}(\hat{\mathbf{n}}^{\mathrm{sca}}, \hat{\mathbf{n}}^{\mathrm{inc}})\mathbf{I}^{\mathrm{inc}}. \tag{21.8}$$

The 4×4 phase matrix can be written in terms of the amplitude matrix elements for single particles [*Mishchenko et al.*, 2002]. All elements of the phase matrix have the dimension of area and are real. As the amplitude matrix, the phase matrix depends on ω^{inc} and ω^{sca} even when ψ^{inc} and/or ψ^{sca} equal 0 or π. In general, all 16 elements of the phase matrix are non-zero, but they can be expressed in terms of only seven independent real numbers. Four elements result from the moduli $|S_{ij}|$ $(i, j = 1, 2)$ and three from the phase-differences between S_{ij}. If the incident beam is unpolarized, i.e., $\mathbf{I}^{\mathrm{inc}} = (I^{\mathrm{inc}}, 0, 0, 0)^T$, the scattered light generally has at least one non-zero Stokes parameter other than intensity:

$$I^{\mathrm{sca}} = Z_{11}I^{\mathrm{inc}}, \tag{21.9}$$
$$Q^{\mathrm{sca}} = Z_{21}I^{\mathrm{inc}}, \tag{21.10}$$
$$U^{\mathrm{sca}} = Z_{31}I^{\mathrm{inc}}, \tag{21.11}$$
$$V^{\mathrm{sca}} = Z_{41}I^{\mathrm{inc}}. \tag{21.12}$$

This is the phenomena is traditionally called “polarization”. The non-zero degree of polarization Equation (22.82) can be written in terms of the phase matrix elements

$$p = \frac{\sqrt{Z_{21}^2 + Z_{31}^2 + Z_{41}^2}}{Z_{11}}. \tag{21.13}$$

21.3.3 Extinction matrix

In the special case of the exact forward direction ($\hat{\mathbf{n}}^{\mathrm{sca}} = \hat{\mathbf{n}}^{\mathrm{inc}}$) the attenuation of the incoming radiation is described by the extinction matrix $\mathbf{K}$. In terms of the Stokes vector we get

$$\mathbf{I}(r\hat{\mathbf{n}}^{\mathrm{inc}})\Delta S = \mathbf{I}^{\mathrm{inc}}\Delta S - \mathbf{K}(\hat{\mathbf{n}}^{\mathrm{inc}})\mathbf{I}^{\mathrm{inc}} + O(r^{-2}). \tag{21.14}$$

Here ΔS is a surface element normal to $\hat{\mathbf{n}}^{\mathrm{inc}}$. The extinction matrix can also be expressed explicitly in terms of the amplitude matrix. It has only seven independent elements. Again the elements depend on ω^{inc} and ω^{sca} even when the incident wave propagates along the z-axis.

21.3.4 Absorption vector

The particle also emits radiation if its temperature T is above zero Kelvin. According to Kirchhoff's law of radiation the emissivity equals the absorptivity of a medium under thermodynamic equilibrium. The energetic and polarization characteristics of the emitted radiation are described by a four-component Stokes emission column vector $\mathbf{a}(\hat{\mathbf{r}}, T, \omega)$. The emission vector is defined in such a way that the net rate, at which the emitted energy crosses a surface element ΔS normal to $\hat{\mathbf{r}}$ at distance r from the particle at frequencies from ω to $\omega + \Delta\omega$, is

$$W^e = \frac{1}{r^2}\mathbf{a}(\hat{\mathbf{r}}, T, \omega)B(T, \omega)\Delta S\Delta\omega, \tag{21.15}$$

where W^e is the power of the emitted radiation and B is the Planck function. In order to calculate $\mathbf{a}$ we assume that the particle is placed inside an opaque cavity of dimensions large compared to the particle and any wavelengths under consideration. We have thermodynamic equilibrium if the cavity and the particle are maintained at the constant temperature T. The emitted radiation inside the cavity is isotropic, homogeneous, and unpolarized. We can represent this radiation as a collection of quasi-monochromatic, unpolarized, incoherent beams propagating in all directions characterized by the Planck blackbody radiation

$$B(T, \omega)\Delta S\Delta\Omega = \frac{\hbar\omega^3}{2\pi^2 c^2\left[\exp\left(\frac{\hbar\omega}{k_B T}\right) - 1\right]}\Delta S\,\Delta\Omega, \tag{21.16}$$

where $\Delta\Omega$ is a small solid angle about any direction, $\hbar$ is the Planck constant divided by 2π, and k_B is the Boltzmann constant. The blackbody Stokes vector is

$$\mathbf{I}_b(T, \omega) = \begin{bmatrix} B(T, \omega) \\ 0 \\ 0 \\ 0 \end{bmatrix}. \tag{21.17}$$

For the Stokes emission vector, which we also call particle absorption vector, we can derive

$$a_i^p(\hat{\mathbf{r}}, T, \omega) = K_{i1}(\hat{\mathbf{r}}, \omega) - \int_{4\pi} \mathrm{d}\hat{\mathbf{r}}' Z_{i1}(\hat{\mathbf{r}}, \hat{\mathbf{r}}', \omega), \quad i = 1, \ldots, 4. \tag{21.18}$$

This relation is a property of the particle only, and it is valid for any particle, in thermodynamic equilibrium or non-equilibrium.

21.3.5 Optical cross sections

The optical cross-sections are defined as follows: The product of the scattering cross section C_{sca} and the incident monochromatic energy flux gives the total monochromatic power removed from the incident wave as a result of scattering into all directions. The product

of the absorption cross section C_{abs} and the incident monochromatic energy flux gives the power which is removed from the incident wave by absorption. The extinction cross section C_{ext} is the sum of scattering and absorption cross section. One can express the extinction cross sections in terms of extinction matrix elements

$$C_{\mathrm{ext}} = \frac{1}{I^{\mathrm{inc}}} (\quad K_{11}(\hat{\mathbf{n}}^{\mathrm{inc}}) I^{\mathrm{inc}} + K_{12}(\hat{\mathbf{n}}^{\mathrm{inc}}) Q^{\mathrm{inc}} + \tag{21.19}$$

$$K_{13}(\hat{\mathbf{n}}^{\mathrm{inc}}) U^{\mathrm{inc}} + K_{14}(\hat{\mathbf{n}}^{\mathrm{inc}}) V^{\mathrm{inc}}), \tag{21.20}$$

and the scattering cross section in terms of phase matrix elements

$$C_{\mathrm{sca}} = \frac{1}{I^{\mathrm{inc}}} \int_{4\pi} \mathrm{d}\hat{\mathbf{r}} (\quad Z_{11}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) I^{\mathrm{inc}} + Z_{12}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) Q^{\mathrm{inc}} + \tag{21.21}$$

$$Z_{13}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) U^{\mathrm{inc}} + Z_{14}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) V^{\mathrm{inc}}). \tag{21.22}$$

The absorption cross section is the difference between extinction and scattering cross section:

$$C_{\mathrm{abs}} = C_{\mathrm{ext}} - C_{\mathrm{sca}}. \tag{21.23}$$

The single scattering albedo ω_0, which is a commonly used quantity in radiative transfer theory, is defined as the ratio of the scattering and the extinction cross section:

$$\omega_0 = \frac{C_{\mathrm{sca}}}{C_{\mathrm{ext}}} \leq 1. \tag{21.24}$$

All cross sections are real-valued positive quantities and have the dimension of area.

The phase function is generally defined as

$$p(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) = \frac{4\pi}{C_{\mathrm{sca}} I^{\mathrm{inc}}} (\quad Z_{11}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) I^{\mathrm{inc}} + Z_{12}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) Q^{\mathrm{inc}} + \tag{21.25}$$

$$Z_{13}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) U^{\mathrm{inc}} + Z_{14}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) V^{\mathrm{inc}}). \tag{21.26}$$

The phase function is dimensionless and normalized:

$$\frac{1}{4\pi} \int_{4\pi} p(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) \, \mathrm{d}\hat{\mathbf{r}} = 1. \tag{21.27}$$

21.4 Scattering, absorption and emission by ensembles of independent particles

The formalism described in the previous chapter applies only for radiation scattered by a single body or a fixed cluster consisting of a limited number of components. In reality, one normally finds situations, where radiation is scattered by a very large group of particles forming a constantly varying spatial configuration. Clouds of ice crystals or water droplets are a good example for such a situation. A particle collection can be treated at each given moment as a fixed cluster, but as a measurement takes a finite amount of time, one measures a statistical average over a large number of different cluster realizations.

Solving the Maxwell equations for a whole cluster, like a collection of particles in a cloud, is computationally too expensive. Fortunately, particles forming a random group can often be considered as independent scatterers. This approximation is valid under the following assumptions:

1. Each particle is in the far-field zone of all other particles.
2. Scattering by the individual particles is incoherent.

As a consequence of assumption 2, the Stokes parameters of the partial waves can be added without regard to the phase. If the particle number density is sufficiently small, the single scattering approximation can be applied. The scattered field in this approach is obtained by summing up the fields generated by the individual particles in response to the external field in isolation from all other particles. If the particle positions are random, one can show, that the phase matrix, the extinction matrix and the absorption vector are obtained by summing up the respective characteristics of all constituent particles.

21.4.1 Single scattering approximation

We consider a volume element containing N particles. We assume that N is sufficiently small, so that the mean distance between the particles is much larger than the incident wavelength and the average particle size. Furthermore we assume that the contribution of the total scattered signal of radiation scattered more than once is negligibly small. This is equivalent to the requirement

$$\frac{N \langle C_{\mathrm{sca}} \rangle}{l^2} \ll 1, \tag{21.28}$$

where $\langle C_{\mathrm{sca}} \rangle$ is the average scattering cross section per particle and l is the linear dimension of the volume element. The electric field scattered by the volume element can be written as the vector sum of the partial scattered fields scattered by the individual particles:

$$\mathbf{E}^{\mathrm{sca}}(\mathbf{r}) = \sum_{n=1}^{N} \mathbf{E_n}^{\mathrm{sca}}(\mathbf{r}). \tag{21.29}$$

As we assume single scattering the partial scattered fields are given according to Equation (21.7):

$$\begin{bmatrix} [E_n^{\mathrm{sca}}(\mathbf{r})]_\psi \\ [E_n^{\mathrm{sca}}(\mathbf{r})]_\omega \end{bmatrix} = \frac{e^{ikr}}{r} \mathbf{S}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) \begin{bmatrix} E_{0\psi}^{\mathrm{inc}} \\ E_{0\omega}^{\mathrm{inc}} \end{bmatrix}, \tag{21.30}$$

where $\mathbf{S}$ is the total amplitude scattering matrix given by:

$$\mathbf{S}(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}) = \sum_{n=1}^{N} e^{i\Delta_n} \mathbf{S}_n(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}}). \tag{21.31}$$

$\mathbf{S}_n(\hat{\mathbf{r}}, \hat{\mathbf{n}}^{\mathrm{inc}})$ are the individual amplitude matrices and the phase Δ_n is given by

$$\Delta_n = k\mathbf{r}_{\mathrm{On}} \cdot (\hat{\mathbf{n}}^{\mathrm{inc}} - \hat{\mathbf{r}}), \tag{21.32}$$

where the vector $\mathbf{r}_{\mathrm{On}}$ connects the origin of the volume element O with the nth particle origin (see Figure 21.2). Since Δ_n vanishes in forward direction and the individual extinction matrices can be written in terms of the individual amplitude matrix elements, the total extinction matrix is given by

$$\mathbf{K} = \sum_{n=1}^{N} \mathbf{K}_n = N \langle \mathbf{K} \rangle, \tag{21.33}$$

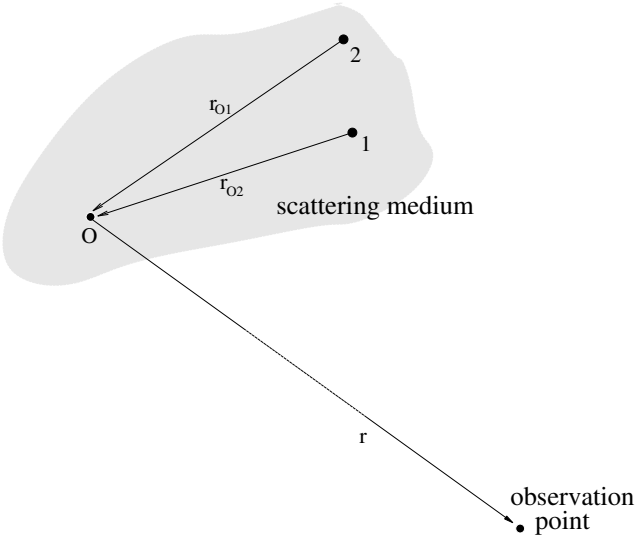

Figure 21.2: A volume element of a scattering medium conststing of a particle ensemble. O is the origin of the volume element, r_{O1} connects the origin with particle 1 and r_{O2} with particle 2. The observation point is assumed to be in the far-field zone of the volume element.

where $\langle \mathbf{K} \rangle$ is the average extinction matrix per particle. One can derive the analog equation for the phase matrix

$$\mathbf{Z} = \sum_{n=1}^{N} \mathbf{Z}_n = N \langle \mathbf{Z} \rangle , \tag{21.34}$$

where $\langle \mathbf{Z} \rangle$ is the average phase matrix per particle. In almost all practical situations, radiation scattered by a collection of independent particles is incoherent, as a minimal displacement of a particle or a slight change in the scattering geometry changes the phase differences entirely. It is important to note, that the ensemble averaged phase matrix and the ensemble averaged extinction matrix have in general 16 independent elements. The relations between the matrix elements, which can be derived for single particles, do not hold for particle ensembles.

21.5 Phenomenological derivation of the radiative transfer equation

When the scattering medium contains a very large number of particles the single scattering approximation is no longer valid. In this case we have to take into account that each particle scatters radiation that has already been scattered by another particle. This means that the radiation leaving the medium has a significant multiple scattered component. The observation point is assumed to be in the far-field zone of each particle, but it is not necessarily in the far-field zone of the scattering medium as a whole. A traditional method in this case is to solve the radiative transfer equation. This approach still assumes, that the particles forming the scattering medium are randomly positioned and widely separated and that the extinction and the phase matrices of each volume element can be obtained by incoherently adding the respective characteristics of the constituent particles. In other words the scattering media is assumed to consist of a large number of discrete, sparsely and randomly distributed particles and is treated as continuous and locally homogeneous. Radiative transfer theory is originally a phenomenological approach based on considering the transport of energy through a medium filled with a large number of particles and ensuring energy conservation. *Mishchenko* [2002] has demonstrated that it can be derived from electromagnetic theory of multiple wave scattering in discrete random media under certain simplifying assumptions.

In the phenomenological radiative transfer theory, the concept of single scattering by individual particles is replaced by the assumption of scattering by a small homogeneous volume element. It is furthermore assumed that the result of scattering is not the transformation of a plane incident wave into a spherical scattered wave, but the transformation of the specific intensity vector, which includes the Stokes vectors from all waves contributing to the electromagnetic radiation field.

The vector radiative transfer equation (VRTE) is

$$\begin{aligned}\frac{\mathrm{d}\mathbf{I}(\mathbf{n},\nu,T)}{\mathrm{d}s} = & -\langle\mathbf{K}(\mathbf{n},\nu,T)\rangle\,\mathbf{I}(\mathbf{n},\nu,T) + \langle\mathbf{a}(\mathbf{n},\nu,T)\rangle\,B(\nu,T) \\ & + \textstyle\int_{4\pi}\mathrm{d}\mathbf{n}'\,\langle\mathbf{Z}(\mathbf{n},\mathbf{n}',\nu,T)\rangle\,\mathbf{I}(\mathbf{n}',\nu,T),\end{aligned} \tag{21.35}$$

where $\mathbf{I}$ is the specific intensity vector, $\langle\mathbf{K}\rangle$ is the ensemble-averaged extinction matrix, $\langle\mathbf{a}\rangle$ is the ensemble-averaged absorption vector, B is the Planck function and $\langle\mathbf{Z}\rangle$ is the ensemble-averaged phase matrix. Furthermore ν is the frequency of the radiation, T is the temperature, $\mathrm{d}s$ is a path-length-element of the propagation path and $\mathbf{n}$ the propagation direction. Equation (21.35) is valid for monochromatic or quasi-monochromatic radiative transfer. We can use this equation for simulating microwave radiative transfer through the atmosphere, as the scattering events do not change the frequency of the radiation.

The four-component specific intensity vector $\mathbf{I} = (I, Q, U, V)^T$ fully describes the radiation and it can directly be associated with the measurements carried out by a radiometer used for remote sensing. For the definition of the components of the specific intensity vector refer to Section 22, where the Stokes components are described. Since the specific intensity vector is a superposition of Stokes vectors, the polarization state of the specific intensity vector can be analysed in the same way as the polarization state of the Stokes vector.

The three terms on the right hand side of Equation (21.35) describe physical processes in an atmosphere containing different particle types and different trace gases. The first term

represents the extinction of radiation traveling through the scattering medium. It is determined by the ensemble averaged extinction coefficient matrix $\langle \mathbf{K} \rangle$. For microwave radiation in cloudy atmospheres, extinction is caused by gaseous absorption, particle absorption and particle scattering. Therefore $\langle \mathbf{K} \rangle$ can be written as a sum of two matrices, the particle extinction matrix $\langle \mathbf{K}^p \rangle$ and the gaseous extinction matrix $\langle \mathbf{K}^g \rangle$:

$$\langle \mathbf{K}(\mathbf{n}, \nu, T) \rangle = \langle \mathbf{K}^p(\mathbf{n}, \nu, T) \rangle + \langle \mathbf{K}^g(\mathbf{n}, \nu, T) \rangle \,. \tag{21.36}$$

The particle extinction matrix is the sum over the individual specific extinction matrices $\langle \mathbf{K}_i^p(\mathbf{n}, \nu, T) \rangle$ of the N different particles types contained in the scattering medium weighted by their particle number densities n_i^p:

$$\langle \mathbf{K}^p(\mathbf{n}, \nu, T) \rangle = \sum_{i=1}^{N} n_i^p \, \langle \mathbf{K}_i^p(\mathbf{n}, \nu, T) \rangle \,. \tag{21.37}$$

A particle distribution, which can include various particle sizes, shapes and orientations, can be represented by a single particle type, since it is possible to derive an ensemble averaged phase matrix $\langle \mathbf{Z}_i \rangle$, an ensemble averaged extinction matrix $\langle \mathbf{K}_i \rangle$ and an ensemble averaged absorption vector $\langle \mathbf{a}_i \rangle$. The gaseous extinction matrix is directly derived from the scalar gas absorption. As there is no polarization due to gas absorption at cloud altitudes, the off-diagonal elements of the gaseous extinction matrix are zero. At very high altitudes above approximately 40 km there is polarization due to the Zeeman effect, mainly due to oxygen molecules. However, in the toposphere and stratosphere molecular scattering can be neglected in the microwave frequency range. Hence the coefficients on the diagonal correspond to the gas absorption coefficient:

$$\left\langle \mathbf{K}_{l,m}^g(\nu, T) \right\rangle = \langle \alpha^g(\nu, T) \rangle \quad \text{if} l = m \tag{21.38}$$

$$0 \quad \text{if} l \neq m. \tag{21.39}$$

where T is the temperature of the atmosphere and $\langle \alpha^g \rangle$ is the total scalar gas absorption coefficient, which is calculated from the individual absorption coefficients of all M trace gases $\alpha_i^g(P, \nu, T)$ and their volume mixing ratios n_i^g as:

$$\langle \alpha^g(\nu, T) \rangle = \sum_{i=1}^{M} n_i^g \alpha_i^g(\nu, T). \tag{21.40}$$

The second term in Equation (21.35) is the thermal source term. It describes thermal emission by gases and particles in the atmosphere. The ensemble averaged absorption vector $\langle \mathbf{a} \rangle$ is

$$\langle \mathbf{a}(\mathbf{n}, \nu, T) \rangle = \langle \mathbf{a}^p(\mathbf{n}, \nu, T) \rangle + \langle \mathbf{a}^g(\nu, T) \rangle \,, \tag{21.41}$$

where $\langle \mathbf{a}^p \rangle$ and $\langle \mathbf{a}^g \rangle$ are the particle absorption vector and the gas absorption vector, respectively. The particle absorption vector is a sum over the individual absorption vectors $\langle \mathbf{a}_i^p \rangle$, again weighted with n_i^p:

$$\langle \mathbf{a}^p(\mathbf{n}, \nu, T) \rangle = \sum_{i=1}^{N} n_i^p \, \langle \mathbf{a}_i^p(\mathbf{n}, \nu, T) \rangle \,. \tag{21.42}$$

The gas absorption vector is simply

$$\langle \mathbf{a}^g(\nu, T) \rangle = \left(\langle \alpha^p(\nu, T) \rangle, 0, 0, 0 \right)^T. \tag{21.43}$$

The last term in Equation (21.35) is the scattering source term. It adds the amount of radiation which is scattered from all directions $\mathbf{n}'$ into the propagation direction $\mathbf{n}$. The ensemble averaged phase matrix $\langle \mathbf{Z} \rangle$ is the sum of the individual phase matrices $\langle \mathbf{Z}_i \rangle$ weighted with n_i^p:

$$\langle \mathbf{Z}(\mathbf{n}, \mathbf{n}', \nu, T) \rangle = \sum_{i=1}^{N} n_i^p \langle \mathbf{Z}_i(\mathbf{n}, \mathbf{n}', \nu, T) \rangle. \tag{21.44}$$

The scalar radiative transfer equation (SRTE)

$$\begin{aligned} \frac{\mathrm{d}I}{\mathrm{d}s}(\mathbf{n}, \nu, T) \quad &= - \langle K_{11}(\mathbf{n}, \nu, T) \rangle \, I(\mathbf{n}, \nu, T) + \langle a_1(\mathbf{n}, \nu, T) \rangle \, B(\nu, T) \\ &\quad + \textstyle\int_{4\pi} \mathrm{d}\mathbf{n}' \, \langle Z_{11}(\mathbf{n}, \mathbf{n}', \nu, T) \rangle \, I(\mathbf{n}', \nu, T) \end{aligned} \tag{21.45}$$

can be used presuming that the radiation field is unpolarized. This approximation is reasonable if the scattering medium consists of spherical or completely randomly oriented particles, where $\langle \mathbf{K}^p \rangle$ is diagonal and only the first element of $\langle \mathbf{a}^p \rangle$ is non-zero.

21.6 Blackbody radiation

All natural bodies with a temperature > 0 K emit thermal radiation. The thermal motion in the matter is translated by collisions to excitations in the molecules. The transition from the excited state to a lower state causes emission of electromagnetic radiation. Depending on the temperature the distribution of the emission will change. An ideal body that absorbs all incoming radiation is called a blackbody and its emittance follows Planck's formula:

$$B(\lambda, T) = \frac{2\pi h c^2}{\lambda^5} \frac{1}{e^{hc/\lambda k_b T} - 1} \tag{21.46}$$

where B is the emitted radiation, λ the wavelength, T the temperature, h the Planck constant, c the speed of light and k_b the Boltzmann constant. Equation 21.46 is shown in Figure 21.3 for some temperatures.

The maximum of the Planck formula is a function temperature and is given by the Wien's displacement law. The maximum of Equation 21.46 is found at:

$$\lambda_{max} = \frac{K}{T} \tag{21.47}$$

where is λ_{max} the wavelength of maximum emittance and $K = 2.898 \cdot 10^{-3}$ Km. Consequently the maximum is encountered at a shorter wavelength for a higher temperature as can be seen in Figure 21.3.

No natural object can be said to be a perfect blackbody but the Sun and the Earth can approximately be treated as blackbodies with temperature of about 6000 and 290 K, respectively, to explain some basic conditions. Wien's displacement law gives maximum for the solar radiation at 0.55 μm while the Earth thermal radiation is maximal around 10 μm. The

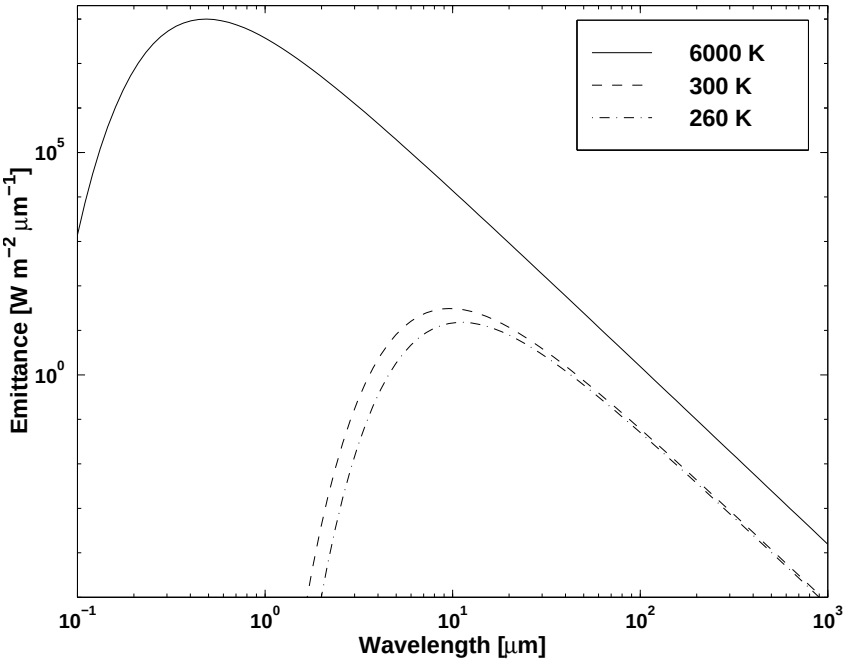

Figure 21.3: The blackbody radiation as a function of wavelength for some temperatures.

solar maximum coincides with a region of high atmospheric transmission and this explains why the evolution has placed the vision between about 400 - 700 nm, the amount of energy at surface level is maximal in this range.

If the radiation from the Sun is scaled to compensate for the attenuation due to the distance it will be found that the radiation from the Earth itself will dominate for wavelengths longer than about 5 μm. This means, for example, that remote sensing techniques in the thermal IR and microwave region primarily detect thermal emission from the surface or the atmosphere while observations in the optical and UV regions use absorbed, scattered or reflected solar radiation. This also explains why gases that exhibit absorption between 5 - 50 μm are called greenhouse gases, they absorb partly some of the outgoing radiation from the surface and the sea.

21.7 Simple solution without scattering and polarization

If scattering can be neglected and the atmosphere is assumed to be in local thermodynamic equilibrium, the radiative transfer equation gets unusually simple. These assumptions will be made below and they are normally valid for the infrared region and longer wavelengths as in the microwave region. For these conditions the atmospheric absorption and emission are linked and the basic problem to determine the radiative transfer is to calculate the absorption. At the wavelengths considered rotational and vibrational transitions are the dominating absorbing processes.

The basic equation describing radiative transfer along a specific direction is

$$\frac{dI(\nu)}{dl} = -k(l,\nu)I(\nu) + S(l,\nu) \tag{21.48}$$

where I is the intensity per unit area, ν the frequency, l the distance along the propagation path, k the total absorption coefficient (summed over all species and transitions) and S the source term. In this general expression both k and S include effects of scattering, i.e. energy lost and gained in the viewing direction due to scattering. The discussion will here be restricted to situations where the scattering can be neglected. This can be a valid assumption for cloud free conditions and wavelengths greater than a few micrometers. At microwave wavelengths scattering can normally be neglected also when clouds are present. For the frequencies considered and altitudes below about 75 km the molecules in the atmosphere can also be considered to be in local thermodynamic equilibrium. This implies that the de-excitation and excitation of a transition have the same probability and the absorption and emission coefficients will be equal. The Kirchoff law then states that the emitted radiance, the source function, is

$$S = k(l,\nu)B(T(l),\nu) \tag{21.49}$$

For the conditions given, no scattering and local thermodynamic equilibrium, the radiative transfer equation is obtained by combining Equations 21.48 and 21.49. The resulting differential equation can be solved to give

$$I(\nu) = I_0(\nu)e^{-\int_0^h k(l',\nu)dl'} + \int_0^h k(l,\nu)B(T(l),\nu)e^{-\int_0^l k(l',\nu)dl'}dl \tag{21.50}$$

where the receiver is assumed to be placed at $l = 0$ and h is the distance along the path to the limit of the media. I_0 is the intensity at the point h which can represent thermal emission from the surface, solar radiation at top of the atmosphere or cosmic background radiation depending on the observation geometry. When discussing radiative transfer the quantity optical depth, τ, is commonly used and it is defined as

$$\tau(l,\nu) = \int_0^l k(l',\nu)dl' \tag{21.51}$$

and Equation 21.50 can be written as

$$I(\nu) = I_0(\nu)e^{-\tau(h,\nu)dl'} + \int_0^h k(l,\nu)B(T(l),\nu)e^{-\tau(l',\nu)}dl \tag{21.52}$$

The terms inside the integral found in this equation have a simple physical meaning, the radiation emitted at one point is $kBdl$ and this quantity is attenuated by the factor $e^{-\tau}$ before it reaches the observation point.

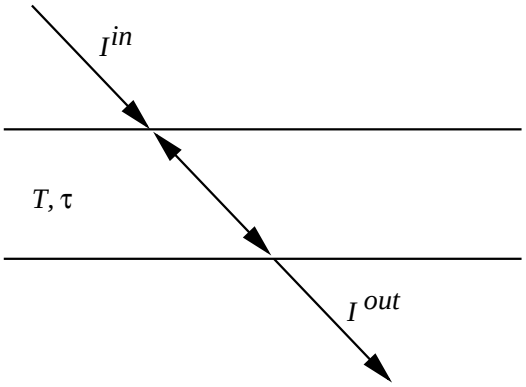

Figure 21.4: Schematic picture of the radiative transfer through a medium with constant temperature.

21.8 Special solutions

If the total emission along the propagation path can be neglected compared to the transmitted part of the incoming radiation, the radiative transfer equation is simplified to the well known Beer-Lambert law:

$$I(\nu) = I_0(\nu)e^{-\tau(h,\nu)} \tag{21.53}$$

This equation can for example be used when evaluating solar occultation observations.

If the temperature is constant through the medium studied (Fig. 21.4) the integral in Equation 21.50 can be solved analytically:

$$I^{out} = I^{in}e^{-\tau} + B(T,\nu)(1 - e^{-\tau}) \tag{21.54}$$

where is τ the total optical thickness of the medium. Two special cases can be distinguished. If the layer is totally optically thick ($\tau \to \infty$) then I^{in} is totally absorbed and $I^{out} = B$, the medium emits as a blackbody. If the layer has no absorption ($\tau = 0$) then Equation 21.54 gives $I^{out} = I^{in}$ as expected.

In microwave radiometry the measured intensity is normally presented by means of the brightness temperature, T_b. This quantity is derived from the Rayleigh-Jeans approximation of the Planck function:

$$B(T,\nu) \approx \frac{2\nu^2 k_b T}{c^2} = \frac{2k_b T}{\lambda^2} \tag{21.55}$$

This equation is valid when $h\nu \ll kT$ which is the case in the microwave region due to the relatively low frequencies. If the temperature is 50 K, hv equals kT at 1.04 THz. The important aspect of Equation 21.55 is the linear relationship between the intensity and the physical temperature. The natural definition of brightness temperature, T_b, is then

$$T_b(\nu) = \frac{\lambda^2}{2k_b T} I(\nu) \tag{21.56}$$

The difference between the brightness temperature and the physical temperature (corresponding to the actual intensity) increases with frequency which is exemplified in Figure

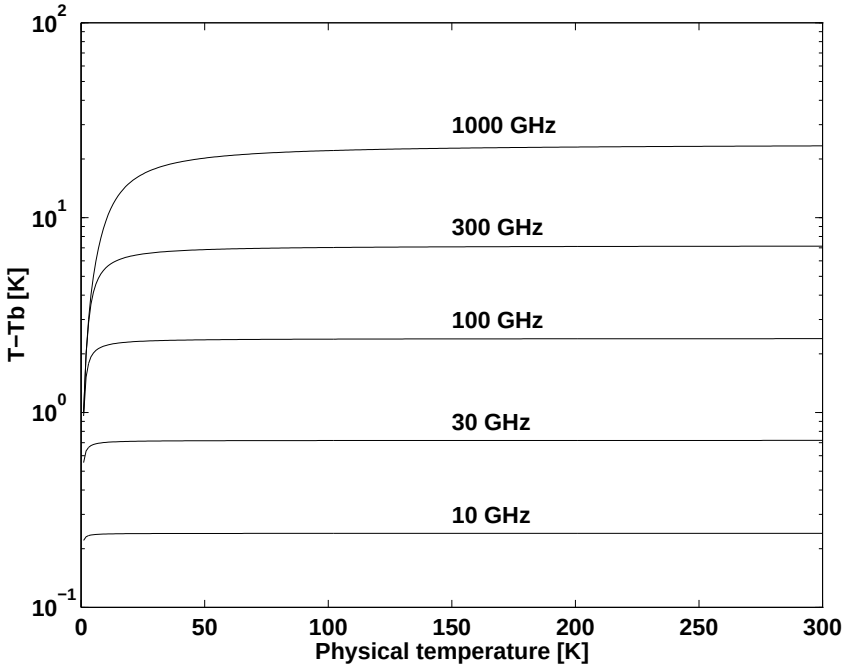

Figure 21.5: The difference between the physical temperarature of a blackbody and the equivalent brightness temperature calculated using the Rayleight-Jeans approximation.

21.5. The differences for higher frequencies are certainly not negligible and the brightness temperature shall not be mistaken for the physical temperature. The important fact is that the brightness temperature has a linear relationship to the intensity and gives a more intuitive understanding of the magnitude of the emission. In the Rayleigh-Jeans limit Equation 21.50 can be written as

$$T_b(\nu) = T_{b0}(\nu)e^{-\tau(h,\nu)dl'} + \int_0^h k(l,\nu)T(l)e^{-\tau(l',\nu)}dl \tag{21.57}$$

Chapter 22

Polarization and Stokes parameters

The present version of ARTS implements the radiative transfer equation in tensor form, i.e., for the 4 components of the Stokes vector, not just for its first component, the intensity or radiance. This means that the model can include polarization dependence in absorption or scattering processes. It is therefore necessary to give some details on the polarization of radiation, the definition of the Stokes parameters, and the definition of antenna polarization.

22.1 Polarization directions

Electromagnetic waves in homogeneous, isotropic media are transverse waves, i.e., their oscillating electric and magnetic fields are in a plane perpendicular to the propagation direction. The choice of two basis vectors – we shall call them polarization directions here – that span that transverse plane is arbitrary; often they are called "horizontal" and "vertical" and correspond to some horizontal and vertical direction of the particular setting. Nevertheless, what is meant by horizontal/vertical, or parallel/perpendicular, is purely a matter of definition.

Here, we stick to the system called laboratory frame or fixed frame, used by Mishchenko or Tsang (FIXME: references): We use a coordinate system where the z-axis points toward local zenith (FIXME: does the x-axis point toward north?). We denote the propagation direction of radiation by a unit vector $\mathbf{n} = \mathbf{k}/k$, where k is the wave number. $\mathbf{n}$ is given by two angles, the zenith angle θ , i.e., the angle between $\mathbf{n}$ and the z-axis, and the azimuth angle ϕ, i.e., the angle between the projection of $\mathbf{n}$ into the xy-plane and the x-axis:

$$\mathbf{n} = \begin{pmatrix} \cos\phi\sin\theta \\ \sin\phi\sin\theta \\ \cos\theta \end{pmatrix} \tag{22.1}$$

Then we define the polarization directions by the partial derivatives of $\mathbf{n}$ with respect to θ and ϕ. We shall call them θ-direction (also: vertical) and ϕ-direction (also: horizontal), respectively, see Figure 22.1. Their unit basis vectors are

---

History

040524 Section scattering matrices by Patrick Eriksson.

040426 Created and written by Christian Melsheimer.

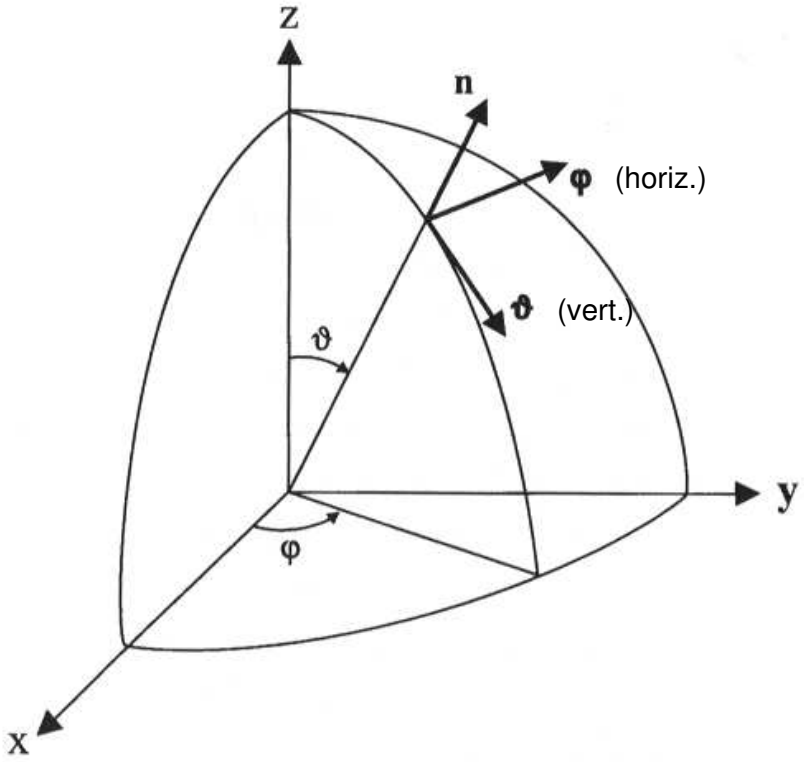

Figure 1 Laboratory coordinate system.

Figure 22.1: The definition of the polarization directions, adapted from Mishchenko FIXME: reference

$$\mathbf{e}_\theta = \mathbf{e}_v = \frac{\partial \mathbf{n}}{\partial \theta} \Big/ \left\| \frac{\partial \mathbf{n}}{\partial \theta} \right\| = \begin{pmatrix} \cos\phi\cos\theta \\ \sin\phi\cos\theta \\ \sin\theta \end{pmatrix} \tag{22.2}$$

$$\mathbf{e}_\phi = \mathbf{e}_h = \frac{\partial \mathbf{n}}{\partial \phi} \Big/ \left\| \frac{\partial \mathbf{n}}{\partial \phi} \right\| = \begin{pmatrix} -\sin\phi \\ \cos\phi \\ 0 \end{pmatrix} \tag{22.3}$$

The vectors $\mathbf{n}$, $\mathbf{e}_\theta$ (=$\mathbf{e}_v$), $\mathbf{e}_\phi$ (=$\mathbf{e}_h$) are mutually orthogonal and define (in the mentioned order) a right-handed system, i.e., $(\mathbf{n} \times \mathbf{e}_\theta) \cdot \mathbf{e}_\phi = 1$ and the same for all cyclic permutations.

22.2 Plane monochromatic waves

Plane monochromatic electromagnetic waves are commonly written in the form

$$\mathbf{E}(\mathbf{x}, t) = \begin{bmatrix} E_v \\ E_h \end{bmatrix} e^{i(\mathbf{kx} - \omega t)} = (E_v \mathbf{e}_v + E_h \mathbf{e}_h)\, e^{i(\mathbf{kx} - \omega t)} \tag{22.4}$$

where $\mathbf{E}$ is the electric field vector, the subscripts v and h denote the components with vertical and horizontal polarization, respectively. E_v and E_h, the amplitudes, are complex numbers, $\mathbf{k}$ and ω are the wavenumber vector and the angular frequency, respectively, of the plane wave, and the unit vectors $\mathbf{e}_v = (1,0)^T$, $\mathbf{e}_h = (0,1)^T$. It is always implicitly understood that the actual, physical, electric field is the real part of the above expression. Rewriting the complex amplitudes E_v and E_h using real, non-negative amplitudes a_v and a_h, and phases δ_v and δ_h,

$$E_v = a_v e^{i\delta_v} \, , E_h = a_h e^{i\delta_h} \tag{22.5}$$

the actual electric field vector $\tilde{\mathbf{E}}$ is

$$\tilde{\mathbf{E}}(\mathbf{x},t) = \mathrm{Re}[\mathbf{E}(\mathbf{x},t)] = \begin{bmatrix} a_v \cdot \cos(\mathbf{kx} - \omega t + \delta_v) \\ a_h \cdot \cos(\mathbf{kx} - \omega t + \delta_h) \end{bmatrix} \tag{22.6}$$

In general, instruments do not measure the electric or magnetic field vectors of an electromagnetic wave, but rather the time-averaged intensity, i.e., the energy flux, F. This is the time-averaged Poynting vector (which, in turn, is proportional to the square of the electric field), thus:

$$\begin{aligned} F &= \sqrt{\frac{\epsilon}{\mu}}\overline{(\tilde{\mathbf{E}}(\mathbf{x},t))^2} \\ &= \sqrt{\frac{\epsilon}{\mu}}\left(a_v^2\overline{\cos^2(\mathbf{kx} - \omega t + \delta_v)} + a_h^2\overline{\cos^2(\mathbf{kx} - \omega t + \delta_h)}\right) \end{aligned} \tag{22.7}$$

The overline denotes the time average which for cosine squares is 1/2, thus:

$$F = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(a_v^2 + a_h^2) \tag{22.8}$$

Taking into account that for plane, monochromatic waves the time average always results in a factor $\frac{1}{2}$, we can also directly write the intensity using the electric field vector in complex notation (Eq. 22.4).

$$\begin{aligned} F &= \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}\mathbf{E}(\mathbf{x},t)\cdot\mathbf{E}^*(\mathbf{x},t) \\ &= \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_v^* + E_h E_h^*) \end{aligned} \tag{22.9}$$

where the asterisk denotes complex conjugation.

In addition to the flux, three more intensity quantities are defined as in the following equations. They are called *Stokes parameters*:

$$I = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_v^* + E_h E_h^*) \tag{22.10}$$

$$Q = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_v^* - E_h E_h^*) \tag{22.11}$$

$$U = -\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_h^* + E_h E_v^*) \tag{22.12}$$

$$V = i\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_h E_v^* - E_v E_h^*) \tag{22.13}$$

Written as a row or column vector, (I,Q,U,V) is called *Stokes vector*. Note that sometimes, S_0, S_1, S_2, S_3 is used instead of I, Q, U, V. Using the amplitude/phase notation

from Eq. (22.5), we can rewrite the Stokes parameters as

$$I = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(a_v^2 + a_h^2) \tag{22.14}$$

$$Q = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(a_v^2 - a_h^2) \tag{22.15}$$

$$U = -\sqrt{\tfrac{\epsilon}{\mu}}a_v a_h \cos(\delta_v - \delta_h) \tag{22.16}$$

$$V = -\sqrt{\tfrac{\epsilon}{\mu}}a_v a_h \sin(\delta_v - \delta_h) \tag{22.17}$$

The Stokes parameters fully characterize the electromagnetic wave and therefore contain the same information as the electric field vector (except for one absolute phase). Since instruments generally measure intensities (fluxes), describing electromagnetic radiation by the Stokes parameters is more practical than describing it by the electric (or magnetic) field vector. Furthermore, the Stokes parameters are always real numbers. Note that the Stokes parameters are sometimes defined with different signs of Q, U, or V (the definitions and signs used here are based on *Mishchenko et al.* [2000]). Moreover, their normalization may vary. In particular, the Stokes parameters can be normalized to represent radiance or irradiance (instead of intensity), which is usually done in radiative transfer contexts.

In order understand what the Stokes parameters mean, we have to go back to the electric field vector and see what polarization state it describes. To do so, we look at the curve that the tip of the physical electric field vector $\tilde{\mathbf{E}}$ describes with time at a fixed position $\mathbf{x_0}$:

$$\tilde{E}_v(t) = a_v \cos(\Delta_v - \omega t) \tag{22.18}$$

$$\tilde{E}_h(t) = a_h \cos(\Delta_h - \omega t) \tag{22.19}$$

where $\Delta_{v,h} = \mathbf{k}\mathbf{x_0} + \delta_{v,h}$. To see that this is an ellipse, we first split the cosines using the addition theorem:

$$\tilde{E}_v(t) = a_v \cos\Delta_v \cos(\omega t) + a_v \sin\Delta_v \sin(\omega t) \tag{22.20}$$

$$\tilde{E}_h(t) = a_h \cos\Delta_h \cos(\omega t) + a_h \sin\Delta_h \sin(\omega t) \tag{22.21}$$

In order to have the tip of $\tilde{\mathbf{E}}$ describe an ellipse with semi-major axis $a_0 \cos\beta$ and semi-minor axis $a_0 \sin\beta$, where $a_0^2 = a_v^2 + a_h^2$, it should have the following form

$$\tilde{E}_v(t) = a_0 \sin\beta \cos(\omega t) \tag{22.22}$$

$$\tilde{E}_h(t) = a_0 \cos\beta \sin(\omega t) \tag{22.23}$$

Here β must be between $-45°$ and $45°$: the tip of the vector $\tilde{\mathbf{E}}$ describes a circle for $\beta = \pm 45°$ (circular polarization), oscillates along the h-axis for $\beta = 0$ (linear polarization) and else describes an ellipse (cf. Figure 22.2). The sense of rotation is counterclockwise for positive β (corresponding to left-circular or left-elliptic polarization) and clockwise for negative β (corresponding to right-circular or right-elliptic polarization). Since $|\tan\beta|$ is the ratio of the semi-minor and semi-major axes of the ellipse (the ellipticity), β is called the ellipticity angle. Note that the semi-major axis is oriented along the positive h-axis. To have the major axis of the ellipse enclose an arbitrary angle ζ ($0 \le \zeta < 180°$) with the h-axis, we apply a rotation matrix and get the equation for an ellipse with arbitrary shape (ellipticity) and orientation (cf. Figure 22.3):

$$\tilde{E}_v(t) = a_0(\sin\beta \cos(\omega t)\cos\zeta + \cos\beta \sin(\omega t)\sin\zeta) \tag{22.24}$$

$$\tilde{E}_h(t) = a_0(-\sin\beta \cos(\omega t)\sin\zeta + \cos\beta \sin(\omega t)\cos\zeta) \tag{22.25}$$

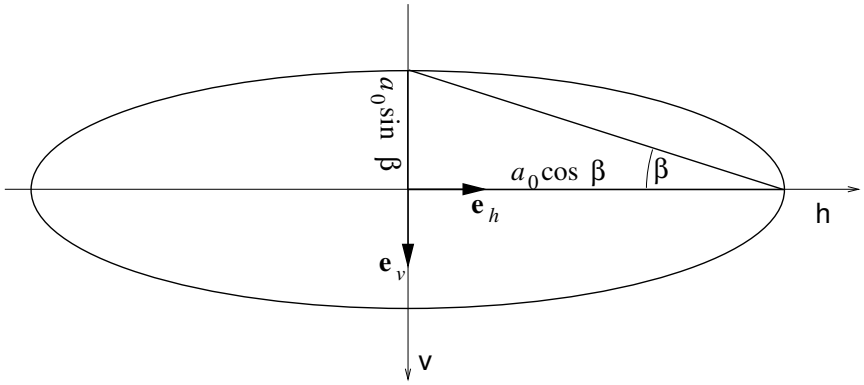

Figure 22.2: The ellipse that the electric field vector describes with time, with the major axis oriented along the h-axis.

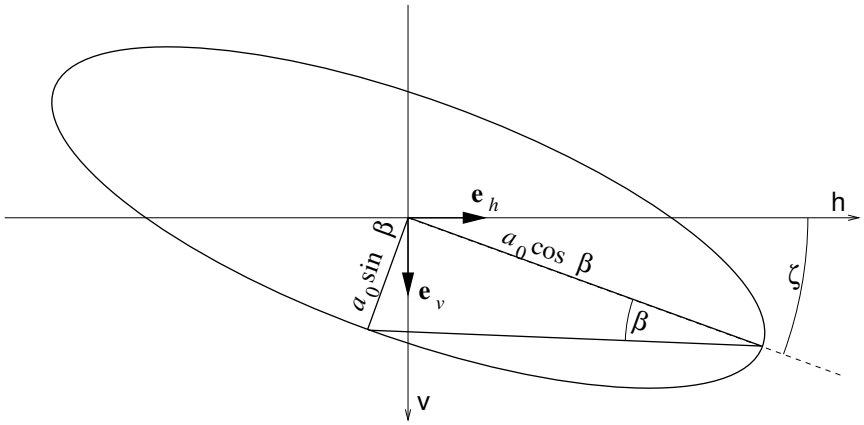

Figure 22.3: The ellipse that the electric field vector describes with time, with the major axis oriented arbitrarily.

With these definitions, horizontal polarization corresponds to $\beta = 0°$ and $\zeta = 0°$; vertical polarization to $\beta = 0°$ and $\zeta = 90°$; left-circular to $\beta = 45°$ and any value of ζ; right-circular to $\beta = -45°$ and any value of ζ.

Now we want to establish a direct connection between the parameters β and ζ describing the shape (ellipticity) and orientation of the polarization ellipse on the one hand, and the amplitudes a_v and a_h and phases δ_v and δ_h of the components of the electric field vector on the other hand. Comparing the $\sin(\omega t)$ and $\cos(\omega t)$ terms in Eqs. (22.24)-(22.25) with the corresponding terms in Eqs. (22.20)-(22.21), we get:

$$a_v \cos \Delta_v = a_0 \sin \beta \cos \zeta \tag{22.26}$$

$$a_v \sin \Delta_v = a_0 \cos \beta \sin \zeta \tag{22.27}$$

and

$$a_h \cos \Delta_h = -a_0 \sin \beta \sin \zeta \tag{22.28}$$

$$a_h \sin \Delta_h = a_0 \cos \beta \cos \zeta \tag{22.29}$$

Multiplying Eq. (22.26) with Eq. (22.28), and Eq. (22.27) with Eq. (22.29) and adding up the results, we get

$$a_v a_h(\cos\Delta_v \cos\Delta_h + \sin\Delta_v \sin\Delta_h) = a_0^2 \sin\zeta\cos\zeta(\cos^2\beta - \sin^2\beta) \tag{22.30}$$

Using the addition theorems for sinusoidals and taking into account that $\Delta_v - \Delta_h = \delta_v - \delta_h$:

$$\frac{a_v a_h}{a_0^2}\cos(\delta_v - \delta_h) = \tfrac{1}{2}\sin(2\zeta)\cos(2\beta) \tag{22.31}$$

In a similar way, subtracting the product of Eq. (22.27) with Eq. (22.28) from the product of Eq. (22.26) with Eq. (22.29) and adding up the results, we get

$$-\frac{a_v a_h}{a_0^2}\sin(\delta_v - \delta_h) = \tfrac{1}{2}\sin(2\beta) \tag{22.32}$$

The above two equations tell us how to translate the amplitudes (a_v, a_h) and phases (δ_v, δ_h) of the vertical and horizontal component of the electric field into the orientation and shape of the ellipse that the tip of the electric field vector describes with time. We can obtain one further relation by subtracting the sum of the squares of Eq. (22.28) and Eq. (22.29) from the sum of the squares of Eq. (22.26) and Eq. (22.27):

$$a_v^2 - a_h^2 = -a_0^2\cos(2\zeta)\cos(2\beta) \tag{22.33}$$

Finally, we use the above 3 equations (22.31), (22.32) and (22.33) to rewrite the Stokes parameters (Eqs. 22.14-22.17) as

$$I = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}a_0^2 \tag{22.34}$$

$$Q = -\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}a_0^2\cos(2\zeta)\cos(2\beta) \tag{22.35}$$

$$U = -\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}a_0^2\sin(2\zeta)\cos(2\beta) \tag{22.36}$$

$$V = -\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}a_0^2\sin(2\beta) \tag{22.37}$$

FIXME: $\beta < 0$ is right-handed pol. (see above, consistent with Jackson and others); thus $V > 0$. This conflicts with Mishchenko's book (p.26).

Thus, we can get the orientation angle ζ of the ellipse from

$$\tan(2\zeta) = \frac{U}{Q} \tag{22.38}$$

Since $0 \le 2\zeta < 360°$, there are 2 solutions for ζ for a given pair U, Q. This ambiguity is resolved by looking at Eq. 22.35, taking into account that $|\beta| \le 45°$ and thus $\cos(2\beta) \ge 0$: The sign of $\cos(2\zeta)$ must be the same as the sign of $-Q$.

We get the ellipticity angle β from

$$\tan(2\beta) = -\frac{V}{(Q^2 + U^2)^{1/2}} \tag{22.39}$$

I is the total intensity of the radiation, Q is the difference in the intensity of the vertically and horizontally polarized components (cf. section 22.3, i.e., next section). I is always non-negative, and Q, U, and V are between $+I$ and $-I$, since they can be expressed as a product

of I with sines and/or cosines (Eqs. 22.35-22.37). Note also that the 4 Stokes parameters are not independent, since the following equality applies:

$$I^2 = Q^2 + U^2 + V^2 \tag{22.40}$$

Some examples of Stokes parameters for specific polarizations are given at the end of the next section (p. 221)

22.3 Measuring Stokes parameters

The three different ways given so far to write the Stokes parameters (Eqs. 22.10ff., Eqs. 22.14ff., Eqs. 22.34ff.) are not very helpful if we actually want to measure the Stokes parameters. So here we are going to rewrite them while keeping in mind that most instruments can just measure intensities of radiation.

We have seen above that the Stokes parameter Q is the difference in the intensity of the vertically and horizontally polarized components (Eq. 22.11, or 22.15)

$$Q = I_v - I_h \tag{22.41}$$

where

$$I_v = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}} E_v E_v^* \tag{22.42}$$

$$I_h = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}} E_h E_h^* \tag{22.43}$$

Thus if we measure I_v and I_h using – for optical wavelengths – a polarizer aligned with the v- and the h-axis, respectively, or using – for microwaves – two appropriately aligned dipole antennas, we can directly obtain I by taking their sum and Q by taking their difference.

U and V can likewise be expressed as differences of intensities, but not with respect to the linear base $\mathbf{e}_v$and $\mathbf{e}_h$:

We recall Eq. (22.4), omitting the oscillatory term:

$$\mathbf{E} = (E_v \mathbf{e}_v + E_h \mathbf{e}_h) \tag{22.44}$$

Now we want to write $\mathbf{E}$ by two components along polarization axes at $\pm 45°$ with respect to the h-axes. The basis vectors are thus (cf. Figure 22.4)

$$\mathbf{e}_{+45°} = \sqrt{\tfrac{1}{2}}\,(\mathbf{e}_h - \mathbf{e}_v) \tag{22.45}$$

$$\mathbf{e}_{-45°} = \sqrt{\tfrac{1}{2}}\,(\mathbf{e}_h + \mathbf{e}_v) \tag{22.46}$$

and we get the field vector in this modified linear basis:

$$\mathbf{E} = \underbrace{\sqrt{\tfrac{1}{2}}\,(E_v + E_h)}_{E_{-45°}} \mathbf{e}_{-45°} + \underbrace{\sqrt{\tfrac{1}{2}}\,(-E_v + E_h)}_{E_{+45°}} \mathbf{e}_{+45°} \tag{22.47}$$

With the definitions of intensities of the components,

$$I_{-45°} = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}} E_{-45°} E_{-45°}^* \tag{22.48}$$

$$I_{+45°} = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}} E_{+45°} E_{+45°}^* \tag{22.49}$$

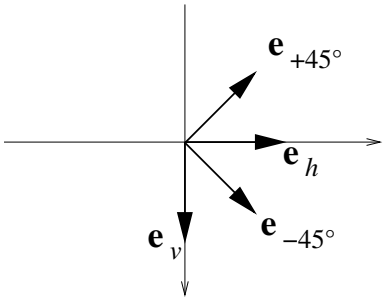

Figure 22.4: Two sets of basis vectors for the linear basis.

we get for their difference:

$$\begin{aligned} I_{-45^\circ} - I_{+45^\circ} &= \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}\left[\tfrac{1}{2}(E_v + E_h)(E_v^* + E_h^*) - \tfrac{1}{2}(-E_v + E_h)(-E_v^* + E_h^*)\right] \\ &= \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_h^* + E_h E_v^*) \end{aligned} \tag{22.50}$$

Therefore (cf. Eq. 22.12)

$$U = I_{+45^\circ} - I_{-45^\circ} \tag{22.51}$$

Thus if we measure I_{+45° and I_{-45° using – for optical wavelengths – a polarizer aligned at $+45^\circ$ and -45° with respect to the h-axis, respectively, or using – for microwaves – two appropriately aligned dipole antennas, we can directly obtain U by taking their difference.

In order to see how to measure the fourth Stokes parameter, V, we have to transform to the circular basis, i.e., express $\mathbf{E}$ by a left-hand (LH) and a right-hand (RH) circularly polarized component. The relevant equations:

Basis vectors

$$\mathbf{e}_{LH} = \sqrt{\tfrac{1}{2}}\,(\mathbf{e}_v + i\mathbf{e}_h) \tag{22.52}$$

$$\mathbf{e}_{RH} = \sqrt{\tfrac{1}{2}}\,(\mathbf{e}_v - i\mathbf{e}_h) \tag{22.53}$$

Field vector in circular base

$$\mathbf{E} = \underbrace{\sqrt{\tfrac{1}{2}}\,(E_v - iE_h)}_{E_{LH}}\,\mathbf{e}_{LH} + \underbrace{\sqrt{\tfrac{1}{2}}\,(E_v + iE_h)}_{E_{RH}}\,\mathbf{e}_{RH} \tag{22.54}$$

Intensity of the components

$$I_{LH} = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}E_{LH}E_{LH}^* \tag{22.55}$$

$$I_{RH} = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}E_{RH}E_{RH}^* \tag{22.56}$$

Their difference

$$\begin{aligned} I_{LH} - I_{RH} &= \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}\left[\tfrac{1}{2}(E_v - iE_h)(E_v^* + iE_h^*) - \tfrac{1}{2}(E_v + iE_h)(E_v^* - iE_h^*)\right] \\ &= i\tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}(E_v E_h^* - E_h E_v^*) \end{aligned} \tag{22.57}$$

Therefore (cf. Eq. 22.13):

$$V = I_{RH} - I_{LH} \tag{22.58}$$

Thus if we measure I_{RH} and I_{LH} using – for microwaves – appropriate helical beam antennas, we can directly obtain V by taking their difference. Unfortunately, for optical wavelengths, we cannot measure I_{RH} and I_{LH} directly with the help of filters like polarizers and retarders[1]. However, a combination of a retarder and a polarizer can be used to measure the sum of I and V:

The light first passes through a retarder that delays the phase of the horizontally polarized component by 90° with respect to the phase of the vertically polarized component (a quarter-wave plate). A phase delay by 90°can be expressed as a multiplication of the horizontal component by i, so the resulting electric field vector $\mathbf{E}'$ is

$$\mathbf{E}' = (E_v \mathbf{e}_v + iE_h \mathbf{e}_h) \tag{22.59}$$

The light then passes through a polarizer that is aligned at $-45°$ with respect to the h-axis. This means we have to project $\mathbf{E}'$ onto $\mathbf{e}_{-45°}$, resulting in

$$\mathbf{E}'' = (\mathbf{E}' \cdot \mathbf{e}_{-45°})\mathbf{e}_{-45°} = \sqrt{\tfrac{1}{2}}\,(E_v + iE_h)\,\mathbf{e}_{-45°} \tag{22.60}$$

Measuring the intensity now, we get

$$\begin{aligned} I'' &= |\mathbf{E}''|^2 \\ &= \tfrac{1}{2}\,(E_v + iE_h)\,(E_v^* - iE_h^*) \\ &= \tfrac{1}{2}\,(|E_v|^2 + |E_h|^2 - i(E_v E_h^* - E_h E_v^*)) \\ &= \tfrac{1}{2}(I + V) \end{aligned} \tag{22.61}$$

Here is a summary of the Stokes parameters in terms of intensities of orthogonal components:

$$I = I_v + I_h = I_{-45°} + I_{+45°} = I_{RH} + I_{LH} \tag{22.62}$$

$$Q = I_v - I_h \tag{22.63}$$

$$U = I_{+45°} - I_{-45°} \tag{22.64}$$

$$V = I_{RH} - I_{LH} \tag{22.65}$$

We see that Q and U are both related to linear polarization, while V is related to circular polarization.

Here are the Stokes parameters for some standard polarizations:

polarization	$(I,\ Q,\ U,\ V)$
horizontal	$(I, -I,\ 0,\ 0)$
vertical	$(I, +I,\ 0,\ 0)$
linear $\pm 45°$	$(I,\ 0, \mp I,\ 0)$
right-circular	$(I,\ 0,\ 0,\ I)$
left-circular	$(I,\ 0,\ 0, -I)$

[1] A retarder allows the phase of two orthogonal components of light to be varied with respect to each other

22.4 Partial polarization

The equality $I^2 = Q^2 + U^2 + V^2$ (Eq. 22.40) is valid for the ideal case of a monochromatic plane wave that is completely polarized, i.e., where the amplitudes a_v and a_h and the phases δ_v and δ_v are fixed and do not vary with time. This means that the plane wave is emitted by one coherent source.

In reality, i.e., in the case of natural radiation, the amplitudes and phases fluctuate, since the radiation originates from several sources that do not emit radiation coherently, and since the emission from one source usually has very short coherence times. This means that we usually have a superposition of radiation from several incoherent sources, and that the polarization state of the radiation from each source fluctuates as well[2]. Typically, such fluctuations have time scales that are longer than the period $(2\pi/\omega)$ of the oscillation, but that are still shorter than the integration time of the instrument that measures the radiation. Thus, the instrument measures an incoherent superposition of time averages over of the fluctuating polarization. If the fluctuations are random for all the sources and if the different sources emit incoherently and are not in any way oriented, then there is no preferred orientation, ellipticity or handedness of the emitted radiation, which is then called unpolarized. This is the case for radiation from the sun. If the fluctuations are not completely random, the radiation is called partially polarized.

To quantify this rather heuristic argumentation, we express the above-mentioned ideas in the language of the Stokes parameters: The Stokes parameters I, Q, U, V derived from measurements result from the superposition of radiation from many sources and/or the average over emission events with individual Stokes parameters I_i, Q_i, U_i, V_i. Since the different sources and/or emission events are incoherent, the Stokes parameters – which are intensity, not amplitude quantities – can simply be added up:

$$I = \sum_i I_i\,,\ Q = \sum_i Q_i\,,\ U = \sum_i U_i\,,\ V = \sum_i V_i \tag{22.66}$$

In the case of unpolarized radiation, i.e., when the amplitudes and phases, or equivalently, the orientation angle ζ and the ellipticity angle β are random (uniformly distributed), Q, U, and V each cancel out.

The equality $I_i^2 = Q_i^2 + U_i^2 + V_i^2$ (cf. Eq. 22.40) still holds for each contribution i, but for the resulting I, Q, U, V, we have in general the inequality

$$I^2 \geq Q^2 + U^2 + V^2 \tag{22.67}$$

To prove it, we must once again go back to the amplitude/phase notation (Eqs. 22.14ff.), also cf. *Chandrasekhar* [1960, chap. I.15], but we shall omit the factor $\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}$ on the right-hand sides, for the sake of better readability:

$$I = \sum_i I_i = \sum_i \left(a_v^{(i)}\right)^2 + \sum_i \left(a_h^{(i)}\right)^2 \tag{22.68}$$

$$Q = \sum_i Q_i = \sum_i \left(a_v^{(i)}\right)^2 - \sum_i \left(a_h^{(i)}\right)^2 \tag{22.69}$$

$$U = \sum_i U_i = -2\sum_i a_v^{(i)} a_h^{(i)} \cos\delta^{(i)} \tag{22.70}$$

[2]This does, of course, not apply to coherent sources like lasers or coherent radars

$$V = \sum_i V_i = 2\sum_i a_v^{(i)} a_h^{(i)} \sin\delta^{(i)} \tag{22.71}$$

$$\tag{22.72}$$

where $\delta^{(i)} = \delta_v^{(i)} - \delta_h^{(i)}$. We get

$$\begin{aligned} I^2 - Q^2 - U^2 - V^2 &= 4\sum_i \left(a_v^{(i)}\right)^2 \sum_i \left(a_h^{(i)}\right)^2 \\ &\quad -4\left(\sum_i a_v^{(i)} a_h^{(i)} \cos\delta^{(i)}\right)^2 \\ &\quad -4\left(\sum_i a_v^{(i)} a_h^{(i)} \sin\delta^{(i)}\right)^2 \end{aligned} \tag{22.73}$$

The first term on the right-hand side can be rearranged as

$$\sum_i \left(a_v^{(i)} a_h^{(i)}\right)^2 + \sum_{\substack{i,j \\ i\neq j}} \left(a_v^{(i)} a_h^{(j)}\right)^2 \tag{22.74}$$

The other two terms can be rearranged similarly to yield:

$$\begin{aligned} &-\sum_i \left(a_v^{(i)} a_h^{(i)}\right)^2 \left[\cos^2\delta^{(i)} + \sin^2\delta^{(i)}\right] \\ &-\sum_{\substack{i,j \\ i\neq j}} a_v^{(i)} a_h^{(i)} a_v^{(j)} a_h^{(j)} \left[\cos\delta^{(i)}\cos\delta^{(j)} + \sin\delta^{(i)}\sin\delta^{(j)}\right] \end{aligned} \tag{22.75}$$

Putting this into Eq. 22.73 (and dividing by 4), the sums over just i cancel and we get:

$$\begin{aligned} (I^2 - Q^2 - U^2 - V^2)/4 &= \sum_{\substack{i,j \\ i\neq j}} \left(a_v^{(i)} a_h^{(j)}\right)^2 \\ &\quad -\sum_{\substack{i,j \\ i\neq j}} a_v^{(i)} a_h^{(i)} a_v^{(j)} a_h^{(j)} \cos(\delta^{(i)} - \delta^{(j)}) \end{aligned} \tag{22.76}$$

where the cosine addition theorem was used. In the summation, we now change from $i \neq j$ to $i < j$, so we have to symmetrize the first term (the second term is already symmetric with respect to i and j and therefore just gets a factor 2):

$$\begin{aligned} (I^2 - Q^2 - U^2 - V^2)/4 &= \sum_{\substack{i,j \\ i<j}} \Big[\left(a_v^{(i)} a_h^{(j)}\right)^2 + \left(a_v^{(j)} a_h^{(i)}\right)^2 \\ &\qquad -2\left(a_v^{(i)} a_h^{(j)}\right)\left(a_v^{(j)} a_h^{(i)}\right)\cos(\delta^{(i)} - \delta^{(j)})\Big] \end{aligned} \tag{22.77}$$

Each summand of the sum on the right-hand side is positive, since it is greater than or equal to $(a_v^{(i)} a_h^{(j)} - a_v^{(j)} a_h^{(i)})^2$, which completes the proof. The right-hand side vanishes only if $\delta^{(i)} = \delta^{(j)}$ and $a_v^{(i)}/a_h^{(i)} = a_v^{(j)}/a_h^{(j)}$ for all i, j, i.e., if the phase difference and amplitude ratio between the horizontal and vertical component of the electric field is the same for all contributions, in other words: if all contributions have the same polarization.

For completeness, we shall now restate the definition of the Stokes component, extended to include natural radiation (i.e., including the case of partially polarized and unpolarized radiation). Instead of summing over the individual emission events, we use ensemble averages, denoted by angular brackets:

$$I = \frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}\langle E_v E_v^* + E_h E_h^* \rangle \tag{22.78}$$

$$Q = \frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}\langle E_v E_v^* - E_h E_h^* \rangle \tag{22.79}$$

$$U = -\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}\langle E_v E_h^* - E_h E_v^* \rangle \tag{22.80}$$

$$V = i\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}\langle E_h E_v^* - E_v E_h^* \rangle \tag{22.81}$$

Except for the ensemble average $\langle .. \rangle$, the definition is identical to the one for monochromatic, plane waves (Eqs. 22.10 to 22.13).The same applies to the second and third definitions of the Stokes parameters (Eqs. 22.14 to 22.17 and Eqs. 22.34 to 22.37, respectively). Note that the fourth definition (Eqs. 22.62 to 22.65) which uses sums and differences of intensities, is equally valid for fully polarized, partially polarized and unpolarized radiation. The definition of intensities, however, has to include the ensemble average: $I_h = \langle E_h E_h^* \rangle$ etc.

Now we can define a measure for the degree of polarization, p, as:

$$p = \frac{\sqrt{Q^2 + U^2 + V^2}}{I} \tag{22.82}$$

For completely polarized radiation, $Q^2 + U^2 + V^2 = I^2$, so $p = 1$, and for unpolarized radiation, $Q = U = V = 0$, so $p = 0$.

Furthermore, it can be convenient to define the the polarized component of radiation by

$$I_p^2 = Q^2 + U^2 + V^2 \tag{22.83}$$

and the unpolarized component as

$$I_u = I - I_p \tag{22.84}$$

Thus, partially polarized radiation, described by a Stokes vector (I, Q, U, V), can be regarded as as a superposition of completely polarized radiation described by the Stokes vector (I_p, Q, U, V) and unpolarized radiation described by the Stokes vector $(I_u, 0, 0, 0)$. We see that the Stokes parameter formalism can conveniently deal with partially polarized and with unpolarized radiation, much in contrast to the formalism using the electric field (amplitude and phase).

In addition to the degree of polarization, p, we can define measures for the circularity and the linearity of the polarization. Recalling Eqs. (22.64) and (22.65), we can define the degree of linear polarization, p_{lin}, as

$$p_{lin} = \frac{\sqrt{Q^2 + U^2}}{I} \tag{22.85}$$

and the the degree of circular polarization, p_{circ}, as

$$p_{circ} = \frac{V}{I} \tag{22.86}$$

22.4.1 Polarization of Radiation in the Atmosphere

The radiation encountered in atmospheric sounding (for which ARTS is intended) is natural radiation, coming from the sun, space (cosmic background), and/or the atmosphere and the Earth surface (thermal radiation, scattered radiation)[3]. Radiation from the sun is unpolarized, as already mentioned; the same applies for the cosmic background. In contrast, radiation emitted by the ground can be weakly polarized, dependent on material, texture and direction. Radiation emitted by the atmosphere (thermal radiation) is almost unpolarized because of the random orientation of the air molecules. An exception might be caused by the Zeeman effect induced in oxygen molecules by the – anisotropic – Earth's magnetic field. Scattering of radiation by oriented particles, e.g. cirrus clouds, is sensitive to polarization, and generally increases the degree of polarization. Typically $I > |Q| > |U|, |V|$.

FIXME: Give some typical numbers for the relative magnitude of I, Q, U, V for realistic radiation in relevant cases ($I \gg Q \gg U, V$ when?)

22.4.2 Antenna polarization

Finally we want to know what an antenna of arbitrary polarization response (antenna polarization) measures if radiation of some other arbitrary polarization is incident on it.

In order to clarify the concept, we first consider some trivial examples: We assume an antenna that receives only vertically polarized radiation.

- If the incident radiation is fully horizontally polarized, the antenna will measure nothing.
- If the incident radiation is fully vertically polarized, the antenna will measure the full intensity of the radiation.
- If the radiation is fully left- or right-circularly polarized, the antenna will measure half of the full intensity, for circularly polarized radiation is made up of equal portions of vertically and horizontally polarized radiation, superimposed with a phase lag of 90°.

In order to be able to describe the general case, we first have to formalize the description of the antenna polarization. Polarized radiation is described by

1. the Jones vector, or
2. the Stokes vector, or
3. intensity, I, orientation angle, ζ (i.e., the angle between the major axis of the polarization ellipse and the horizontal polarization direction), and ellipticity angle, β (see p. 216).

Since the intensity of the radiation is the absolute square (the squared "length") of the complex Jones vector, or, in other words, the first Stokes component, I, the polarization alone is defined by

1. a normalized Jones vector, or

[3]This is not so for active sounding techniques that use a coherent source, such as lidar.

2. three normalized Stokes components Q, U, and V (where $Q^2 + U^2 + V^2 = 1$), or
3. the orientation angle ζ and the ellipticity angle β (see Eq. 22.38 to 22.39).

In the same way, the polarization of the antenna can be described in one of three ways:

1. a normalized Jones vector

$$\mathbf{e} = \begin{bmatrix} e_v \\ e_h \end{bmatrix} \qquad \text{where} \quad \mathbf{e} \cdot \mathbf{e}^* = 1 \tag{22.87}$$

   (note that in the scalar product of two complex vectors, the second one has to be complex-conjugated.)

2. a normalized Stokes vector

$$\mathbf{i} = (1, q, u, v) \qquad \text{where} \quad q^2 + u^2 + v^2 = 1 \tag{22.88}$$

3. the two angles ζ and β. According to Eq. 22.34 to 22.37, we have:

$$q = -\cos(2\zeta)\cos(2\beta) \tag{22.89}$$
$$u = -\sin(2\zeta)\cos(2\beta) \tag{22.90}$$
$$v = -\sin(2\beta) \tag{22.91}$$

Now we can calculate the intensity I' the antenna measures. In terms of the electrical fields, i.e., Jones vectors, we just have to project the Jones vector $\mathbf{E}$ of the incident radiation onto the normalized Jones vector $\mathbf{e}$ of the antenna,

$$\mathbf{E}' = (\mathbf{E} \cdot \mathbf{e}^*)\mathbf{e} \tag{22.92}$$

(this is in effect like passing through a polarizer) and then take its absolute square

$$I' = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}\,|\mathbf{E}'|^2 = \tfrac{1}{2}\sqrt{\tfrac{\epsilon}{\mu}}\,|(\mathbf{E} \cdot \mathbf{e}^*)|^2 \tag{22.93}$$

With some elementary algebra (mainly using that $\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}E_v E_v^* = (I + Q)/2$, $\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}E_h E_h^* = (I - Q)/2$, $\frac{1}{2}\sqrt{\frac{\epsilon}{\mu}}E_v E_h^* = -(U - \mathrm{i}V)/2$ which follow immediately from Eq. 22.10 to 22.13) this can be rewritten in terms of the of the Stokes vector $\mathbf{I}$ of the incident radiation and the Stokes vector $\mathbf{i}$ of the antenna. It turns out to be just a scalar product:

$$I' = \frac{1}{2}\mathbf{i} \cdot \mathbf{I} \tag{22.94}$$

22.5 The scattering amplitude matrix

The electric field, $[E_v, E_h]^T$, originating from a single scattering event of an incident electric field $[E_v^0, E_h^0]^T$ may in the far field be written as (c.f. Eq. 21.7)

$$\begin{bmatrix} E_v \\ E_h \end{bmatrix} = f(r) \begin{bmatrix} S_2 & S_3 \\ S_4 & S_1 \end{bmatrix} \begin{bmatrix} E_v^0 \\ E_h^0 \end{bmatrix}, \tag{22.95}$$

where S_j are the scattering amplitude functions and all distance effects are put into the function $f(r)$. Using Stokes based nomenclature, the equation above becomes

$$\begin{bmatrix} I \\ Q \\ U \\ V \end{bmatrix} = g(r)\mathbf{F} \begin{bmatrix} I^0 \\ Q^0 \\ U^0 \\ V^0 \end{bmatrix}, \tag{22.96}$$

where all distance effects are put into the function $g(r)$ and the transformation matrix $\mathbf{F}$ can be expressed as [*Liou*, 2002, Sec. 5.4.3].

$$\mathbf{F} = \begin{bmatrix} \frac{1}{2}(M_2+M_3+M_4+M_1) & \frac{1}{2}(M_2-M_3+M_4-M_1) & S_{23}+S_{41} & -D_{23}-D_{41} \\ \frac{1}{2}(M_2+M_3-M_4-M_1) & \frac{1}{2}(M_2-M_3-M_4+M_1) & S_{23}-S_{41} & -D_{23}+D_{41} \\ S_{24}+S_{31} & S_{24}-S_{31} & S_{21}+S_{34} & -D_{21}+D_{34} \\ D_{24}+D_{31} & D_{24}-D_{31} & D_{21}+D_{34} & S_{21}-S_{34} \end{bmatrix}. \tag{22.97}$$

The elements of $\mathbf{F}$ are finally given by the following expressions:

$$M_k = |S_k|^2, \tag{22.98}$$

$$S_{kj} = S_{jk} = (S_j S_k^* + S_k S_j^*)/2, \tag{22.99}$$

$$-D_{kj} = D_{jk} = i(S_j S_k^* - S_k S_j^*)/2, \qquad j,k = 1,2,3,4. \tag{22.100}$$

Depending on the properties of the scattering event, the structure of the matrix $\mathbf{F}$ differs. Two special cases are:

$$S_1 = S2, \quad S_3 = S_4 = 0 \quad \rightarrow \quad \mathbf{F} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & x & 0 & 0 \\ 0 & 0 & x & 0 \\ 0 & 0 & 0 & x \end{bmatrix}, \tag{22.101}$$

$$S_3 = S_4 = 0 \quad \rightarrow \quad \mathbf{F} = \begin{bmatrix} x & x & 0 & 0 \\ x & x & 0 & 0 \\ 0 & 0 & x & x \\ 0 & 0 & x & x \end{bmatrix}, \tag{22.102}$$

where x indicates elements deviating from 0. Many (most?) natural materials have the property that S_4 is the complex conjugate of S_3 ($S_3 = S_4^*$) and this results in that $\mathbf{F}$ is a symmetric matrix (in general with all element positions filled).

Part VI

Bibliography and Appendices

Bibliography

Anderson, G. P., S. A. Clough, F. X. Kneizys, J. H. Chetwynd, and E. P. Shettle, AFGL atmospheric constituent profiles (0–120 km), *Tech. Rep. TR-86-0110*, AFGL, 1986.

Balluch, M., and D. Lary, Refraction and atmospheric photochemistry, *J. of Geophys. Res.*, *102*, 8845–8854, 1997.

Bohren, C., and D. R. Huffman, *Absorption and Scattering of Light by Small Particles*, Wiley Science Paperback Series, 1998.

Chandrasekhar, S., *Radiative Transfer*, Dover Publications, New York, 1960.

Czekala, H., Microwave radiative transfer calculations with multiple scattering by non-spherical hydrometeors, Ph.D. thesis, Rheinische Friedrich-Wilhems-Universitt Bonn, Auf dem Hgel 20, 53121 Bonn, 1999.

Davis, C. P., C. Emde, and R. S. Harwood, A 3d polarized reversed monte carlo radiative transfer model for mm and sub-mm passive remote sensing in cloudy atmospheres, *IEEE Transactions on Geoscience and Remote Sensing*, *submitted*, 2004.

Degl'Innocenti, E. L., and M. L. Degl'Innocenti, *Solar Physics*, *97*, 239, 1985.

Emde, C., A polarized discrete ordinate scatterig model for radiative transfer simulations in spherical atmospheres with thermal source, Ph.D. thesis, University of Bremen, 2005.

Emde, C., and T. R. Sreerekha, Development of a RT model for frequencies between 200 and 1000 GHz, WP1.2 Model Review, *Tech. rep.*, ESTEC Contract No AO/1-4320/03/NL/FF, 2004.

Emde, C., S. A. Buehler, C. Davis, P. Eriksson, T. R. Sreerekha, and C. Teichmann, A polarized discrete ordinate scattering model for simulations of limb and nadir longwave measurements in 1D/3D spherical atmospheres, *J. of Geophys. Res.*, *109*, 2004.

Eriksson, P., Microwave radiometric observations of the middle atmosphere: Simulations and inversions, Ph.D. thesis, School of Electrical and Computer Engineering, Chalmers University of Technology, Sweden, 1999.

Eriksson, P., and F. Merino, On simulating passive observations of the middle atmosphere in the range 1 - 1000 GHz, *Tech. Rep. 179*, Department of Radio and Space Science, Chalmers University of Technology, Sweden, 1997.

Eriksson, P., F. Merino, D. Murtagh, P. Baron, P. Ricaud, and J. de la Nöe, Studies for the Odin sub-millimetre radiometer: 1. Radiative transfer and instrument simulation, *to appear in Canadian Journal of Physics*, 2000.

Eriksson, P., M. Ekström, S. Bühler, and C. Melzheimer, Efficient forward modelling by matrix representation of sensor responses, *International Journal of Remote Sensing*, *27*, 1793–1808, 2006.

Evans, K. F., S. J. Walter, A. J. Heymsfield, and M. N. Deeter, Modeling of submillimeter passive remote sensing of cirrus clouds, *J. Appl. Met.*, *37*, 184–205, 1998.

Golub, G. H., and C. F. V. Loan, *Matrix computations*, Johns Hopkins series in the mathematical sciences ; 3, 2nd ed., Hopkins Univ. Press, 1991.

Hochstadt, H., *Differential Equations: A Modern Approach*, Holt, Rinehart, and Winston, 1964.

Ishimoto, H., and K. Masuda, A monte carlo approach for the calculation of polarized light: application to an incident narrow beam, *Journal of Quantitative Spectroscopy and Radiative Transfer*, *72*, 462–483, 2002.

Jackson, J. D., *Classical electrodynamics*, John Wiley & Sons, New York, 1998.

Kyle, T., *Atmospheric transmission, emission and scaterring*, Pergamon Press, 1991.

Liou, K. N., *An introduction to atmospheric radiation*, 2nd ed., Academic Press, 2002.

Liu, J. S., *Monte Carlo Strategies in Scientific Computing*, Springer-Verlag, 2001.

Liu, Q., C. Simmer, and E. Ruprecht, Three-dimensional radiative transfer effects of clouds in the microwave spectral range, *J. of Geophys. Res.*, *101*, 4289–4298, 1996.

McFarquhar, G. M., and A. J. Heymsfield, Parametritation of tropical cirrus ice crystal size distributions and implications for radiative transfer: Results from CEPEX, *Journal of the Atmospheric Sciences*, *54*, 2187–2200, 1997.

Mishchenko, M. I., Calculation of the amplitude matrix for a nonspherical particle in a fixed orientation, *Appl. Opt.*, pp. 1026–1031, 2000.

Mishchenko, M. I., Vector radiative transfer equation for arbitrarily shaped and arbitrarily oriented particles: a microphysical derivation from statistical electromagnetics, *Applied Optics*, *41*, 7114–7134, 2002.

Mishchenko, M. I., and L. D. Travis, 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*, 309–324, 1998.

Mishchenko, M. I., J. W. Hovenier, and L. D. Travis, eds., *Light Scattering by Nonspherical Particles*, Academic Press, 2000, ISBN 0-12-498660-9.

Mishchenko, M. I., L. D. Travis, and A. A. Lacis, *Scattering, Absorption and Emission of Light by Small Particles*, Cambridge University Press, 2002, ISBN 0-521-78252.

Moler, C. B., and C. F. V. Loan, Nineteen dubious ways to compute the exponential of a matrix, *SIAM Review*, *20*, 801–836, 1979.

Montenbruck, O., and E. Gill, *Satellite orbits: Models, methods and applications*, Springer Verlag, 2000.

Oikarinen, L., E. Sihvola, and E. Kyrola, Multiple scattering radiance in limb-viewing geometry, *J. of Geophys. Res.*, *104*, 31261–31274, 1999.

Press, W., S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical recipes in C*, 2nd ed., Cambridge University Press, 1997.

Roberti, L., and C. Kummerow, Monte carlo calculations of polarized microwave radiation emerging from cloud structures, *J. of Geophys. Res.*, *104*, 2093–2104, 1999.

Rodgers, C., *Inverse methods for atmospheric sounding: Theory and practise*, 1st ed., World Scientific Publishing, 2000.

Rodgers, C. D., Characterization and error analysis of profiles retrieved from remote sounding measurements, *J. of Geophys. Res.*, *95*, 5587–5595, 1990.

Rosenkranz, P. W., Absorption of microwaves by atmospheric gases, in *Atmospheric remote sensing by microwave radiometry*, edited by M. A. Janssen, pp. 37–90, John Wiley & Sons, Inc., 1993, `ftp://mesa.mit.edu/phil/lbl_rt`.

Rothman, L. S., et al., The HITRAN molecular spectroscopic database and HAWKS (HITRAN atmospheric workstation): 1996 edition, *Journal of Quantitative Spectroscopy and Radiative Transfer*, *60*, 665–710, 1998.

Ulaby, F., R. Moore, and A. Fung, *Microwave remote sensing: Active and passive, Volume I: Microwave remote sensing fundamentals and radiometry*, Addison-Wesley Publishing Company, 1981, ISBN 0-201-10759-7 (v. 1).

van de Hulst, H., *Light Scattering by Small particles*, Dover Publications, New York, 1957, corrected republication 1981.

Part VII

Index

Index