ARTS Developer Guide

edited by

Stefan Buehler and Patrick Eriksson

November 23, 2023
ARTS Version 2.5.13 (git: edd714d2)

The content and usage of ARTS are not only described by this document. An overview of ARTS documentation and help features is given in *ARTS User Guide*, Section 1.2. For continuous reports on changes of the source code and this user guide, subscribe to the ARTS developers mailing list at https://www.radiativetransfer.org/contact/.

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

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 https://www.radiativetransfer.org//docs/.

Copyright (C) 2000-2015
Stefan Buehler <sbuehler (at) uni-hamburg.de>
Patrick Eriksson <patrick.eriksson (at) chalmers.se>

The ARTS program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with the program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

Contributing authors

Author/email	Main contribution(s)
Stefan Buehler[a] sbuehler (at) uni-hamburg.de	Editor, Sections 1, 2, 3 and 5.
Mattias Ekström[b]	Section 3.4.
Claudia Emde[c] claudia.emde (at) dlr.de	Section 7.
Patrick Eriksson[b] patrick.eriksson (at) chalmers.se	Editor.
Oliver Lemke[a] olemke (at) core-dump.info	Section 4.
Sreerekha T.R.[c]	Section 6.

The present address is given for active contributors, while for others the address to the institute where the work was performed is given:

[a] Meteorological Institute, University of Hamburg, Bundesstr. 55, 20146 Hamburg, Germany.

[b] Department of Space, Earth and Environment, Chalmers University of Technology, SE-41296 Gothenburg, Sweden.

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

Contents

Chapter 1

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 save you a lot of work to understand how things are organized.**

1.1 Organization

ARTS is written in C++ and uses the cross-platform, open-source build system CMake (`http://www.cmake.org/`). 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, The document that you are reading right now, the ARTS Developer Guide, is located in `doc/uguide`.

There are two different versions of the ARTS package: The development version and the end-user version. Both contain the complete source code, the only difference is that the developers version is where active development takes place.

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-C++ compiler, and the CMake utility. Please see files `arts/README` and `arts/INSTALL` for installation instructions. We are aiming to support recent version of the GNU and clang C++ compilers.

1.2 The ARTS build system

As mentioned above, CMake is used to build the ARTS package. A good introduction to the CMake system can be found in:

`http://www.cmake.org/cmake/project/about.html`

History	
020425	Stefan Buehler: Put this part back in the AUG. Updated.
000728	Stefan Buehler: Added stuff about build system and howto cut a release.
000615	Created by Stefan Buehler.

1.2.1 Configure options

For development, it is recommended to build ARTS using the `RelWithDebInfo` configuration (see below), which aims to provide a reasonable trade-off between debugging capability and performance. To use ARTS in production, however, it is important to perform a release build, which is therefore set as the default configuration.

`-DCMAKE_BUILD_TYPE=RelWithDebInfo`: This is the build option that should be used for development, as it will make it easier to track down errors in the code. It does, however, not disable all compiler optimization, so as to still provide reasonable performance.

`-DCMAKE_BUILD_TYPE=Release`: 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,...).

`-DCMAKE_BUILD_TYPE=Debug`: This switch turns off all optimizations. This should only be used if the `RelWithDebInfo` configuration makes debugging a given problem difficult.

`-DNO_OPENMP=1`: Disables the generation of multi-threaded code. CMake tries to detect if the compiler supports OpenMP and enables it by default.

1.3 Coding conventions

With the aim of improving quality and consistency of the code, all new code that is added to ARTS should adhere to naming and formatting conventions from Google's C++ programming guidelines (https://google.github.io/styleguide/cppguide.html#Formatting). Adhering to a well-defined coding style will make it much easier for your fellow ARTS developers to understand and work with your code. A brief summary of the most important programming style and formatting conventions is given below.

1.3.1 Naming conventions

Naming things is one of the two hard problems in computer science. Certainly, there is no single best way to do it and even with the best names code can still be bad. Yet still, the consistently naming of objects in your code will make it much easier for other developers to read and understand your code.

In general, the use descriptive names in all your code is recommended. Try to avoid abbreviations except when they are very common. Giving proper names to objects increases the readability of the code and decreases the need for explanatory comments.

Variables

Variable names should use lower-case letters. Words should be separated using underscores:

```
Index element_index = 0;
Numeric t_surface = 0.0; // common abbreviation
```

Classes, structs and type names

Classes, structs and user-defined type names should start with a capital letter and use camel case to separate different words.

```
class CovarianceMatrix {};
```

Class member variables

Variables that are defined as data members of classes should be suffixed with a underscore (_). This convention has the important advantage that it allows inferring the scope of variables used inside definitions of member functions.

```
class CovarianceMatrix {
 private:
  Numeric *elements_;
};
```

Constant names

Names of constants should be prefixed with a lower-case k. Following words should use camel case starting with a capital letter.

```
const Numeric kAlmostPi = 3.0;
```

Function names

Function names should start with a capital letter and words should be separated using camel case.

```
void InterpolateTo(const Vector &x_new) {
  ...
}
```

1.3.2 Formatting

The formatting, i.e. the layout of your code, should adhere strictly to the Google guidelines. Google's indentation style is also supported by the clang-format tool, which provides functionality for automatic formatting. A format file for clang-format is provided in ARTS' top-level directory. Most development environments provide support for clang-format, which makes following these guidelines extremely easy.

- Line length of 80 characters
- No tabs, only spaces
- 2 spaces per indentation level
- Opening brace { of function/class/struct definition and if/for blocks on the same line
- No spaces on the inner side of the parentheses of the conditional expressions of if statements.

1.3.3 ARTS-specific rules

Numeric types

Never use `float` or `double` explicitly, use the type `Numeric` instead. This is set by CMake (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 `cmake` with the options `-DINDEX=long` or `-DNUMERIC=double`:

```
cmake -DINDEX=int --DNUMERIC=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.

1.3.4 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.

1.3.5 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.

1.3.6 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.

1.3.7 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.)

1.3.8 Version numbers

The package version number is set in the `VERSION` file in the top level ARTS directory. It will be incremented by the ARTS maintainers when new features or bug fixes have been added to ARTS, not on every commit. Never change this number when working in your own branch. The major and/or minor version number will be incremented on public releases. The micro version indicates the addition of new features during development or bugfixes for stable releases.

1.3.9 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.

1.3.10 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.

Doxygen supports several different comment block styles. Over the years, probably all of them were used somewhere in ARTS. New code should follow the Doxygen JavaDoc style. If you edit existing documentation, it is recommended to convert it to the current style.

File comment

Each header and source code file should contain a doxygen comment stating the original author, creation date and a short summary of its contents.

```
/**
 * @file   dummy.cc
 * @author John Doe <john.doe (at) example.com>
 * @date   2011-03-02
 *
 * @brief  A dummy file.
 *
 * This file has no purpose at all,
 * it just servers as an example...
 */
```

Function comment

Each function should be preceeded with a doxygen comment. It starts with a brief description, ending with a dot. Then follows a more detailed description of the function's purpose and parameters.

The doxygen comment block should be put above the *declaration* of the function, i.e., in the `.h` file. If a function is only declared in a `.cc` file, the comment should be put there instead.

If arguments are modified by the function you should add '[out]' after the `@param` command, just like for the parameter `a` in the example below. If a parameter is both input and output, it should say '[in,out]'. Parameters that are not modified inside the function, e.g. passed by value or const reference, should carry an '[in]'. 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[out]     a This parameter is initialized by the function.
 * @param[in,out]  b This parameter is modified by the function.
 * @param[in]      c This parameter is not changed by the function.
 *
 * @return   Dummy value computed from a and b.
 */
int dummy(int& a, int& b, int c);
```

Classes and structs

Classes und structs must be preceeded by a doxygen comment describing their intent and purpose. A short description should be provided for each member variable. Member function are documented as described in the previous section.

```
/** Brief description of Foobar.
 *
```

```
 * Long description of Foobar.
 */
class FooBar {
 private:
  /** Number of elements. */
  Index nelem;
}
```

Doxymacs for Emacs

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 "JavaDoc")

(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 macros to insert certain types of doxygen comment by name:

- `doxymacs-insert-file-comment`
- `doxymacs-insert-function-comment`
- `doxymacs-insert-blank-multiline-comment`
- `doxymacs-insert-blank-singleline-comment`

1.4 Extending ARTS

1.4.1 How to add a workspace variable

You should read Section 2.2 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!

1.4.2 How to add a workspace variable group

You should read Section 2.2 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. Add the group to the python interface's main module. See the python interface description for adding new groups. Failing to add the group here will likely lead to the inability to import the pyarts module to python. Note that there are several automatic

tests in the pytest suite to ensure that all workspace variables follow a common logic, such as offering XML IO and being picklable, so you must ensure you are adding all of these.

5. That's it! (But as stated above, use this feature wisely)

1.4.3 How to add a workspace method

You should read Section 2.3 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!

1.4.4 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/CMakeLists.txt`. This file states which source files are needed for arts. In the usual case, you just have to add your `.cc` file to the list of source files of the artscore library. Header files are not added to this list.

3. Go to `src` and run: `git add <my_file>` to make your file known to git.

1.4.5 How to add a test case

1. Tests are located in subdirectories in the `controlfiles` folder. Instrument specific test cases are in the `controlfiles/instruments` folder, all other cases are located in the `controlfiles/artscomponents` folder. Create a new subdirectory in the appropriate folder. 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. `controlfiles/artscomponents/doit/TestDOIT.arts`.

   If the subdirectory contains more than one test controlfile, append a short descriptive text to the end of the filename like `controlfiles/artscomponents/montecarlo/TestMonteCarloGaussian.arts`.

3. Copy all required input files into the subdirectory. Input data that is shared among several test cases should be placed in `controlfiles/testdata`.

4. Add an entry for your test case in `controlfiles/CMakeLists.txt`.

1.4.6 How to add a particle size distribution

In `m_psd.cc`, add a workspace method `psdPsdName`, where `PsdName` stands for the name or name tag of the new particle size distribution (PSD) parametrization. (see Section 1.4.3 for details). If several `psdPsdName` methods are based on the same algorithm, add the algorithm as an internal function to `psd.cc`.

1.5 Version control

ARTS uses git for version control. The central or *upstream* repository is hosted on github (`https://github.com/atmtools/arts`). Contributions to ARTS are handled via pull requests on github. This is the de-facto standard workflow for open source development, so the time required to get familiar with it is certainly a worthy investment. Contributing a new feature or bug-fix to ARTS thus involves the following steps:

1. Fork the upstream repository
2. Clone the ARTS fork from *your github account* onto your local workstation
3. Implement your changes
4. Commit and push your changes to your ARTS fork
5. Issue a pull request to merge your ARTS fork with the central repository
6. One of the ARTS core developers will review your code and help with the final integration

The required steps are described below in more detail. Note that all of these steps are so common to modern software development, that it is very easy to find tutorials and manuals online that described them in more detail.

1.5.1 Forking the central repository

The forking of the central repository is through the web interface of `github.com`. This will create a copy of the central repository in *your account*. This repository is your personal version of the ARTS repository. You can change all you want here without breaking ARTS for anyone else.

1.5.2 Clone your ARTS fork

So far your fork of ARTS exists only in your github account somewhere in the *cloud*. To really work with the code, you need to obtain a copy of the code on your local workstation. This is done by *cloning* the repository from your account:

```
git clone https://github.com/<your_username>/arts
```

where `your_username` is the name of your github account. The important point to note here is that you did not clone the central ARTS repository from `github.com/atmtools` but the one from your account. As explained above, this is your personal version of the ARTS repository and you can do anything you want with it.

1.5.3 Update your fork

One consequences of the forking of the central ARTS repository is that the fork in your own github account will not automatically be updated when changes are made to the upstream repository. Before checking in your changes into your repository it is therefore important that you update your repository with the most recent changes made to the upstream repository. For this two steps are required:

1. Register the upstream repository as remote repository for your local clone of your ARTS fork:

   ```
   git remote --add upstream https://github.com/atmtools/arts
   ```

2. Rebase your code onto the newest changes from the upstream repository:

   ```
   git fetch upstream master
   git rebase -i upstream/master
   ```

The first step here needs to be performed only the first time you are going through this process. Afterwards only the second step is required. A detailed description of the rebase step can be found at `https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase`.

1.5.4 Commit and push your changes

After implementing your changes, commit and push them to your ARTS fork. The changes are now publicly available from your github account. Note that at this point you can already share your code with others or across different machines. It has, however, not yet been integrated into the upstream repository. That means your changes are not yet affecting regular users of the ARTS development version.

1.5.5 Issuing a pull request

Once your changes have been pushed to the ARTS repository in your github account, you can issue a pull request. This is done most easily through the github web interface. This will notify the ARTS core developers that you want to integrate your changes into the upstream repository. On issuing your pull request, the ARTS test-suite will be run on your code. If all tests pass, an ARTS core developer will review your code and finally merge it into the central repository.

1.6 Debugging (use of ARTS_ASSERT)

The idea behind ARTS_ASSERT is simple. It exists to stop program execution at times when the precondition or post-condition of the function is not met. 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 ARTS_ASSERT it with a statement like this:

```
ARTS_ASSERT(var1 == var2);
```

or

```
ARTS_ASSERT(var1 == var2, "(",var1,", ",var2,")")
```

In general ARTS_ASSERT takes as argument a boolean expression followed by any number of arguments that the developer wants to pass along to the output operator in case the assertion is invoked. If the boolean expression is true, execution continues. Otherwise a C++ std::runtime_error 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 ARTS_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. Instead, assertions are turned on and off with the global NDEBUG preprocessor macro, which is set or unset automatically by the cmake build configuration. Assertions are enabled in the default cmake build configuration (-DCMAKE_BUILD_TYPE=RelWithDebInfo). They are turned off in the release configuration (-DCMAKE_BUILD_TYPE=Release) and in native configuration (-DCMAKE_BUILD_TYPE=Native).

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.

1.7 Debugging (use of ARTS_USER_ERROR_IF)

The `ARTS_USER_ERROR_IF` command should be used whenever the validity of the program being able to produce expected output is compromised by direct user input. This helps the user debug their controlfiles, python scripts, or C++ API usage. Developing ARTS, you should use these devotedly to ensure that the user input is correct and can be understood by all internal functions called by a user-facing method. Invocation of `ARTS_USER_ERROR_IF` is not a bug in ARTS per se but an indication that some calculations will fail at a later stage given the user input.

The interface is exactly the same as for `ARTS_ASSERT`, however the expectation is that the first boolean argument is false for the program execution to stop. As for the assertions, you can pass any number of arguments to the output operator, however, as opposed to the assertions, you are not allowed to pass no arguments because the user needs to be able to understand what s/he did wrong.

The `ARTS_USER_ERROR_IF` command is also implemented as a macro and can be turned off manually by passing `-DNO_USER_ERRORS=1` to cmake. Note that this is not recommended and that the kinds of errors that it can cause to ARTS execution is beyond the responsibility of the ARTS developers and that we do not even try to understand what might happen. It might speed up execution times, however, so it is provided for users that need all possible speed-ups.

Chapter 2

The workspace

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.

2.1 Implementation files

The 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_workspace.h`:
  Generated from `groups.cc`.
- `auto_md.h`, `auto_md.cc`:
  Generated from `auto_workspace.h`, `agendas.cc`, `groups.cc`, and `methods.cc`.

History

110622 Updated by Oliver Lemke.
020605 Created by Stefan Buehler.

This is achieved by a set of simple C++ programs:

- `make_auto_workspace_h.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`:
  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`.

2.2 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`.

See Section 1.4 for explicit instructions how to add a new WSV to ARTS.

2.3 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_raw.push_back
  ( MdRecord
    ( NAME( "r_geoidSpherical" ),
      DESCRIPTION
      (
       "Sets the geoid to be a perfect sphere.\n"
       "\n"
       "The radius of the sphere is selected by the generic argument r.\n"
      ),
      AUTHORS( "Patrick Eriksson" ),
      OUT( "r_geoid" ),
      GOUT(),
      GOUT_TYPE(),
      GOUT_DESC(),
      IN( "atmosphere_dim", "lat_grid", "lon_grid" ),
      GIN( "r" ),
```

```
      GIN_TYPE(    "Numeric" ),
      GIN_DEFAULT( NODEF     ),
      GIN_DESC( "Radius of the geoid sphere."),
      ));
```

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 the generic output arguments must belong (see below).
6. The *GOUT_DESC*, a list of short descriptions for the generic outputs.
7. 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.
8. The *GIN*, a list of descriptive names for the generic inputs.
9. The *GIN_TYPE*. This is a list of WSV group names. This defines the group to which the generic input arguments must belong.
10. 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.
11. The *GIN_DESC*, a list of short descriptions for the generic inputs.

2.3.1 Specific WSMs

```
md_data_raw.push_back
  ( MdRecord
    ( NAME( "p_gridFromGasAbsLookup" ),
      DESCRIPTION
      (
       "Sets *p_grid* to the frequency grid of *abs_lookup*.\n"
      ),
      AUTHORS( "Patrick Eriksson" ),
      OUT( "p_grid" ),
      GOUT(),
      GOUT_TYPE(),
      GOUT_DESC(),
      IN( "abs_lookup" ),
      GIN(),
      GIN_TYPE(),
      GIN_DEFAULT(),
      GIN_DESC()
      ));
```

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 p_grid, using the WSV abs_lookup as input.

To call this method in the controlfile, you just have to write p_gridFromGasAbsLookup.

2.3.2 Generic WSMs

This class of WSMs 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 vector and sets all elements to the specified value.\n"
       "\n"
       "The vector length is determined by *nelem*.\n"
       ),
      AUTHORS( "Patrick Eriksson" ),
      OUT(),
      GOUT(       "v"       ),
      GOUT_TYPE( "Vector" ),
      GOUT_DESC( "Variable to initialize." ),
      IN( "nelem" ),
      GIN(          "value"    ),
      GIN_TYPE(     "Numeric" ),
      GIN_DEFAULT( NODEF      ),
      GIN_DESC(     "Vector value." )
      ));
```

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 1.2.3 in *ARTS User Guide* for information on the built-in documentation.)

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)
```

2.3.3 Agenda WSMs

2.4 Agendas

2.4.1 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 agendas 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 iyEmissionStandard 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( blackbody\_radiation\_agenda ){
  blackbody\_radiationPlanck
}
```

Chapter 3

Vectors, matrices, tensors, and arrays

This section describes how arbitrary-rank ”tensor” classes are implemented in ARTS and how their objects can be used. Furthermore it describes how arrays of arbitrary type can be constructed and used.

3.1 Implementation files

The arbitrary-rank ”tensor” classes are implemented in the `src/matpack/` folder. The implementation is done through template programming, meaning that the logic for how to use the objects is similar even though the rank may change. Common for all of these types though is that the rank of the object has to be known at compile time.

There are two template classes that represents objects owning their own data. The template class that knows both its rank and exact size at compile time is implemented in `matpack_constexpr.h`. The template class that only knows its rank at compile time but has to allocate memory at runtime is implemented in `matpack_data.h`.

There are also two template classes representing a view of the data. The template class that views a piece of data with known rank and size is implemented also in `matpack_constexpr.h`. The template class that views data of known rank but unknown size at compile time is implemented in `matpack_view.h`.

All four of these will be described more below.

The template class Array (also described below) is implemented in the file `array.h`.

The Sparse class is described in the file `matpack_sparse.h`.

3.2 Arbitrary rank ”tensor” — matpack

This section describes ranked data in ARTS. We call template classes of our arbitrary data template class with rank above 2 ”Tensor”, though we fully admit that they do not conform to all the ideas that a mathematical tensor would. The intent is that they should, but we are

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.

limited by the use that we have had of them, so further extensions that come at no cost to the quality of the current implementation is highly welcome. We use the term ”Vector” to describe a rank 1 realization of the template class. We use the term ”Matrix” to describe a rank 2 realization of the template class. These rank 1 and 2 objects do more closely follow the mathematical description of vectors and matrices, such as supporting the dot-product and matrix-vector, and matrix-matrix multiplications.

As the most common underlying type to perform computations on in ARTS is Numeric, the rank 1 class representing Numeric is called Vector and the rank 2 class is called Matrix. We do not name types above rank 7. Ranked classes representing Numeric between rank 3 and rank 7 are called Tensor3, Tensor4, Tensor5, Tensor6, and Tensor7, respectively. These are all the named types, each of which have their sizes known only at runtime. It is important to be aware that re-interpolating a Tensor7 requires a rank 14 object, however, so the information in this section also applies to versions of the class that are only named internally.

Under the hood, the implementation for these classes is based on C++23 `std::mdspan` and the C++26 proposal for `std::submdspan`. The intent is that you should be able to interact with these objects as if they were actually of the template class `std::mdspan`. Note that neither of these C++ standards are actually available as of yet (nb., 2023-02-28), so the details under the hood relies on the Kokkos implementation (which represents what got accepted for C++23 and what is proposed for C++26).

The underlying data type when the size of the data is known at compile time is `std::array` and the underlying data type when the size of the data is known only at runtime is `std::vector`. Again, it is our intent that we should expose as much of these the underlying data type's behavior in the interface to our own classes as possible.

In short there are two intents of these codes:

1. Represent any data that we might need in ARTS in a concise, fast, and mathematically meaningful way.

2. Offer a functional interface that matches the C++ standard interface as closely as possible.

3.2.1 Defining the template classes

These template classes can own data or view data. The size of the data can be known at compile time or not. This subsection goes through the common type and their used names inside ARTS.

matpack_constant_data — owning and constant size

The data owning template class for the type that knows its size at compile time is called

```
template <typename T, Index... alldim>
struct matpack_constant_data;
```

It will hold a type `T` that can be any type that can be held by `std::array` and it will have a rank `N` that is as many Index that you define in the place of `alldim`. Here are a few examples to help you understand:

```
// An object "a" that holds 5 Numeric as a vector:
matpack_constant_data<Numeric, 5> a;
```

```
// An object "a2" that holds 1,2,3,4,5 as Numeric as a vector:
matpack_constant_data<Numeric, 5> a2{1,2,3,4,5};

// An object "b" that holds 6 Index as a 3x2 matrix:
matpack_constant_data<Index, 3, 2> b;

// An object "b" that holds 1,2;3,4;5,6 Index as a 3x2 matrix:
matpack_constant_data<Index, 3, 2> b{1,2,3,4,5,6};

// An object "c" that holds 120 Complex as a 5x4x3x2 tensor 4:
matpack_constant_data<Complex, 5, 4, 3, 2> c;
```

Note that while the name might be confusing, the ”constant” part here refers to ”constant size”. The ”const” and ”constexpr” properties of C++ still has to be applied to make the data actually not mutable.

These are the named versions of these types inside ARTS:

- THERE ARE NO TYPES EXPLICITLY NAMED YET

matpack_constant_view — non-owning and constant size

The template class that does not own its data but points at a compile time constant size of data is called

```
template <typename T, bool constant, Index... alldim>
struct matpack_constant_view;
```

Both `T` and `alldim` are the same as above. The new boolean argument `constant` is held to tell the type if it can mutate its data or not. Generally, you should try to not construct a type like this manually but instead rely on access-patterns to a higher or same rank `matpack_constant_data` or `matpack_constant_view` to generate the type as need be. You can do so by using the access or call operators provided by `matpack_constant_data`. Note also that you can always create a `constant==true` object if you have a `constant==false` object, but that this is irreversible within the type system.

These are the named versions of these types inside ARTS:

- THERE ARE NO TYPES EXPLICITLY NAMED YET

matpack_data — owning and runtime size

The data owning template class that does not know its size but its rank at compile time is called

```
template <typename T, Index N> class matpack_data;
```

Unlike the other data owning object, the size of the data is invariant in most operations by keeping the underlying allocated data constant. This has to be dealt with on a function-by-function basis to keep the rest of the class working. As for the constant data type, the value type will be a `T` — that is possible to be held by a `std::vector` — but the rank is defined explicitly as `N`. Rank 0 objects are not allowed. The sizes of the object has to at some point be given during runtime. Here are a few examples to help you understand:

```
// An object "a" as a rank 1 vector of unknown size:
matpackt_data<Numeric, 1> a;

// An object "a2" as a rank 1 vector of current size 5:
matpackt_data<Numeric, 1> a2(5);

// An object "a3" as a rank 1 vector of current size n:
extern Index n;
matpackt_data<Numeric, 1> a3(n);

// An object "b" that holds 6 Index as a 3x2 matrix:
matpack_data<Index, 2> b(3, 2);

// An object "b2" that holds 6 Index as a 3x2 matrix all of value 1:
matpack_data<Index, 2> b2(3, 2, 1);

// An object "c" that holds 120 Complex as a 5x4x3x2 tensor 4:
matpack_data<Complex, 4> c(5, 4, 3, 2);
```

Note that the same number of Index as the rank of the object has to be given to the constructor for the created object to have an immediate size. If an additional argument is given, it is converted to type `T` and set as the value for all of the elements of the tensor. Since the data is only known at compile time, these objects can all be resized calling the member function `resize`. From the example above, calling `a.resize(5);` gives the object the size 5.

These are the named versions of these types inside ARTS:

Name	Value Type `T`	Rank `N`
Vector	Numeric	1
Matrix	Numeric	2
Tensor3	Numeric	3
Tensor4	Numeric	4
Tensor5	Numeric	5
Tensor6	Numeric	6
Tensor7	Numeric	7
`ComplexVector`	`Complex`	1
`ComplexMatrix`	`Complex`	2

matpack_view — non-owning and runtime size

Likewise, the template class that does not own its data but points to runtime allocated set of data is called

```
template <typename T, Index N, bool constant, bool strided>
class matpack_view;
```

All of `T`, `N` and `constant` are as defined above but because of both legacy and practical reasons, we need an additional boolean argument here: `strided`. Sometimes the memory layout in hardware of data is not *contiguous* or *exhaustive*, that is you cannot expect that as long as you are within the total size of the object, that the next data point is always offset by the same step size. A good example of a strided vector is all the real values from a complex-valued vector; standard implementations of complex values guarantees that

the real value is next to the imaginary value in memory, so the next real value is atleast 2 steps away. In analogy for `matpack_constant_view`, the recommended way to create a `matpack_view` is same or higher ranks of `matpack_data` or `matpack_view`. We also allow viewing a single `T` as the underlying via explicit conversion. Also as for `matpack_constant_view`, you can generate a constant view from a non-constant but never the reverse. The same is true for strided views: you are able to generate a strided view from a exhaustive view but never a exhaustive view from a strided view. Be aware that a `matpack_data` type is neither constant nor strided, which is why these properties are left to the type system and not to the type itself.

These are the named versions of these types inside ARTS:

Name	Value Type `T`	Rank `N`	`constant`	`strided`
`VectorView`	Numeric	1	false	true
`MatrixView`	Numeric	2	false	true
`Tensor3View`	Numeric	3	false	true
`Tensor4View`	Numeric	4	false	true
`Tensor5View`	Numeric	5	false	true
`Tensor6View`	Numeric	6	false	true
`Tensor7View`	Numeric	7	false	true
`ComplexVectorView`	`Complex`	1	false	true
`ComplexMatrixView`	`Complex`	2	false	true
`ConstVectorView`	Numeric	1	true	true
`ConstMatrixView`	Numeric	2	true	true
`ConstTensor3View`	Numeric	3	true	true
`ConstTensor4View`	Numeric	4	true	true
`ConstTensor5View`	Numeric	5	true	true
`ConstTensor6View`	Numeric	6	true	true
`ConstTensor7View`	Numeric	7	true	true
`ConstComplexVectorView`	`Complex`	1	true	true
`ConstComplexMatrixView`	`Complex`	2	true	true
`ExhaustiveVectorView`	Numeric	1	false	false
`ExhaustiveMatrixView`	Numeric	2	false	false
`ExhaustiveTensor3View`	Numeric	3	false	false
`ExhaustiveTensor4View`	Numeric	4	false	false
`ExhaustiveTensor5View`	Numeric	5	false	false
`ExhaustiveTensor6View`	Numeric	6	false	false
`ExhaustiveTensor7View`	Numeric	7	false	false
`ExhaustiveComplexVectorView`	`Complex`	1	false	false
`ExhaustiveComplexMatrixView`	`Complex`	2	false	false
`ExhaustiveComplexTensor3View`	`Complex`	3	false	false
`ExhaustiveConstVectorView`	Numeric	1	true	false
`ExhaustiveConstMatrixView`	Numeric	2	true	false
`ExhaustiveConstTensor3View`	Numeric	3	true	false
`ExhaustiveConstTensor4View`	Numeric	4	true	false
`ExhaustiveConstTensor5View`	Numeric	5	true	false
`ExhaustiveConstTensor6View`	Numeric	6	true	false
`ExhaustiveConstTensor7View`	Numeric	7	true	false
`ExhaustiveConstComplexVectorView`	`Complex`	1	true	false
`ExhaustiveConstComplexMatrixView`	`Complex`	2	true	false
`ExhaustiveConstComplexTensor3View`	`Complex`	3	true	false

3.2.2 Access operations of the template classes

The access patterns we allow currently are through these types

- Index – an integer, e.g., ”row 5”
- `Range` – a strided range of values, e.g., ”row 1 until row 5, every other item”

- `Joker` – an entire dimension, e.g., ”all rows”

Since the constant size data and view types cannot be strided, they make use only of Index and `Joker`. The runtime size data make use of all three types. If the access operation would logically reduce the rank to 0, instead of returning a rank 0 object, something of the underlying value type `T` is returned. Generally, this is a copy of the value if the object is constant or a reference to the value if the object is mutable. Otherwise, if the access operation does not completely reduce the rank of the object, a view of the remaining rank is returned. Don't worry, there are examples below.

Constant size access

There are only two ways to access constant data:

```
// "a" from earlier example:
matpack_constant_data<Numeric, 5> a;
a[0] = 3;  // Set the first element of "a" to three
a(1) = 4;  // Set the second element of "a" to four

// "b" from earlier example
matpack_constant_data<Index, 3, 2> b;
b(0, 0) = 3;  // Set row 0 col 0 of b to 3
b(0, 1) = 4;  // Set row 0 col 1 of b to 4

// c is now a matpack_constant_view<Index, false, 2>
auto c = b(1, joker);

// Set the second element of "c" to five. b(1, 1) is also 5.
c[1] = 5;

// We can only access from the left to remain exhaustive so
// we can just skip all the jokers and do
auto d = b[2];
// So that the type of "d" is the same as the type of "c"

const auto e = a;  // Create a constant of "a"

// We can do the same accessing as before to create a
// matpack_constant_view<Index, true, 2>
auto f = e[1];

// But if we try to write to the data we get a compile-error
// The error is difficult to predict, but it will either say
// that writing to a constant is not allowed or that the
// requirements of "not constant" is not fulfilled
f[0] = 2;  // compilation error
```

The same access pattern works for the template class that owns and the template class that does not own the view of the data

Runtime size access

The access operators for the runtime sized data can be any combination of the three accessing types above or just the square-bracket to access the inner-view. Note that the way you

access an object may change the type returned in subtle ways. Examples:

```
// For a Vector
Vector a(5);
a[joker] = Vector(5, 1);  // Set all of "a" to 1
a[1] = 3;  // Set the second element of "a" to 3
a[Range(2, 2, 1)] = 4;  // Set the 3rd+4th element of "a" to 4

// What are the return types?
a[0];  // This is a Numeric!
a[joker];  // This is an ExhaustiveVectorView!
a[Range(2, 2, 1)];  // This is a VectorView!

// For a Matrix:
Matrix b(3, 2);
b(0, 0) = 10;  // Sets b at row 0 and col 0 to 10
b[1] = 3;  // Set b(1, 0) and b(1, 1) to 3
b(joker, 1) = 2;  // Set b(0, 1), b(1, 1), and b(2, 1) to 2

// What are the return types?
b(0, 0);  // This is a Numeric!
b(joker, joker);  // This is an ExhaustiveMatrixView!
b(1, joker);
b[1];  // These are ExhaustiveVectorView!
b(joker, 0);  // This is a VectorView!
b(1, Range(0, 1));  // This is a VectorView!
```

These access concepts are extendible the any rank. Note that the last example contains exhaustive data layout but the returned type is not exhaustive. That is because any access with range can only be determined at runtime if it is strided or not, but the interface is strict with regards to types in C++.

It is up to the developer to ensure that you never access out-of-bounds elements. To get the extent of a dimension we have these member functions

- `extent(i)` — get the size of the i:th left-most dimension, e.g., rows for $i = 1$ on a matrix
- `nelem` — get the size of a rank 1 tensor
- `ncols` — get the size of the right-most dimension of a rank 2 or higher tensor
- `nrows` — get the size of the right-most bar 1 dimension of a rank 2 or higher tensor
- `npages` — get the size of the right-most bar 2 dimension of a rank 3 or higher tensor
- `nbooks` — get the size of the right-most bar 3 dimension of a rank 4 or higher tensor
- `nshelves` — get the size of the right-most bar 4 dimension of a rank 5 or higher tensor
- `nvitrines` — get the size of the right-most bar 5 dimension of a rank 6 or higher tensor
- `nlibraries` — get the size of the right-most bar 6 dimension of a rank 7 or higher tensor

- `shape` — get the sizes of all dimensions of a rank `N` tensor as a `N`-long array of indices

To get the stride of a given dimension, we offer the member function `stride(i)` for the stride of dimension `i` or `strides` to get the all the strides of a rank `N` tensor as a `N`-long array of indices.

3.2.3 Iterate over elements for faster execution time

All template classes in this part of matpack gives 2 very important and very different types of iterators: elementwise and subviews. This allows us to write composable code that works for any and all types. For instance

```
Vector a({1, 2, 3, 4, 5});  // "a" is {1, 2, 3, 4, 5}
for (auto&& x: a) {
  x = sin(x);  // update x to the sin of itself
}  // "a" is {sin(1), sin(2), sin(3), sin(4), sin(5)}
```

Note that we have a shorter way of writing this that also could generate better runtime:

```
Vector a({1, 2, 3, 4, 5});  // "a" is {1, 2, 3, 4, 5}
matpack::transform(a, sin, a);
```

You can read the implementation of transform in `matpack_math.h`. Here the input and output is the same, and both are Vector, but this works for any types as long as their total size is the same. Even if the two types have a different rank...

For matrices, this works a little bit different:

```
Matrix b(3, 2, 1); // A matrix of {1, 1; 1, 1; 1, 1}
for (auto&& a: b) {
  a[0] = 3;  // Set the first element of "a" to 3
}  // Will now have a matrix of {3, 1; 3, 1;, 3; 1}
```

This happens because the `begin` and `end` iterator pairs are dereferenced to a lower rank view. This works for any higher rank tensor as well, so a tensor of rank 5 views the pair of iterators as a tensor view of rank 4, for example. Remember that constness and stridedness is preserved or decayed-towards in these operations, so the iterators of a constant view can only ever be constant and the iterator of a strided view can only ever return a strided view when dereferenced. But iterators of non-constant and non-strided can be dereferenced to constant or strided views.

Note also that sometimes we just want to do something with all of the elements. For rank 1 tensors, this is the same, but rank 2 and higher ranked tensors behave differently. This is actually the trick to `transform` above, we have a pair of iterators `elem_begin` and `elem_end` that loops over the individual elements of the tensor. Several functions and members of the template classes make use of this feature. Again, the recommendation is to see `matpack_math.h` as it uses many of these iterators for solving various simple transformation and reduction problems (e.g., `sum`, `min`, `transform`, etc...).

The subsection also claims that you get faster execution times using these iterators. This is of course necessary to test on a case-by-case basis. The main benefit we have found in our testing is for exhaustive data and views. In several settings, the compiler seems to emit SIMD code for these exhaustive types. Depending on your operating system and compiler settings, this means some operations may be 2-4 times faster for exhaustive objects when

operated upon by element-wise iterator logic. If you use index access or view iterators some additional calculations to get to the internal value element is needed, which might prevent such SIMD optimizations.

3.2.4 Mathematical and logical operations

All of the normal element-wise mathematical operations are available for objects of the same rank and shape as well as for scalar operations:

```
Vector a({1,2,3,4,5});
a += 1;  // a = {2,3,4,5,6}
a -= 2;  // a = {0,1,2,3,4}
a *= 3;  // a = {0,3,6,9,12}
a /= 2;  // a = {0,1.5,3,4.5,6}

Matrix b(2, 2, 1);  // b = {1,1;1,1}
Matrix c(2, 2, 2);  // c = {2,2;2,2}
b += b;  // b = {2,2;2,2}
c *= c;  // c = {4,4;4,4}
b /= c;  // b = {0.5,0.5;0.5,0.5}
c -= b;  // c = {3.5,3.5;3.5,3.5}
```

We also offer more linear algebra implementations in the `lin_alg.h`, `lin_alg.cc`, `matpack_math.h`, and `matpack_eigen.h` files. The first two of these files support operations such as LU decomposition, solving of linear systems, matrix inversions and diagonalizations, matrix exponents, and similar. The `matpack_math.h` file contains matrix-vector, matrix-matrix, and dot-product like features as well as a few statistical functions. The `matpack_eigen.h` file wraps vectors and matrices with the Eigen3 library so that more direct math works. Be careful using these convenience functions as they can slow down code that would be faster if the direct LAPACK-call is used.

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
```

3.2.5 Notes on minimizing runtime overhead

This is just a list of reminders that anyone coming back to read this should be aware of. There is no specific order to these recommendations, and they are probably hardware bound, so take these with a grain of salt but look at the list as a ”best practices” example.

Exhaustive or not? During the implementation of the modern iteration of `matpack`, we found that exhaustive views of the data often perform between 2-5 times faster than strided views. The cost is of course that you have to implement the function twice if you even end up having to call it with a strided view. Your choice!

3.3 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.

3.3.1 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;
Array<String> a{"ARTS",
                "is",
                "great"}; // Creates an array of String
                          // with these 3 elements.
```

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.

3.3.2 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.

3.4 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, ($nnz \leq 0.5 \times (ncols \times nrows - (ncols + 1))$).

3.4.1 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.
```

3.4.2 What you can do with a Sparse

All examples below assume that `a` is a Sparse.

Identity matrix:

```
a.resize(10,10);
id_mat(a);
```

This sets a to be the identity matrix of size 10×10 (10 rows and 10 columns). Using this function is much faster than setting the diagonal elements to one by yourself. Note that a must be a square matrix.

Resize:

```
a.resize(5,10);
```

This makes a a 5×10 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;
```

The copying of matrices is implemented as deep copy. That means that the complete object is duplicated including all elements in the matrix. The resulting matrices are completely independent of each other, but depending on a this may require considerable amount time and memory.

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 addition and subtraction:

The sums and differences of sparse matrices **with the same dimensions** can be computed as follows:

```
Sparse b(a.nrows(),a.ncols());
Sparse c(a.nrows(),a.ncols());

add( c, a, b ); // c = a + b
a += b;         // a = a + b

sub( c, a, b ); // c = a - b
a -= b;         // a = a - b
```

Scaling of sparse matrices:

Sparse matrices can be scaled by scalar factors as follows:

```
a *= 2.0;  // a = 2.0 * a
a /= 2.0;  // a = 0.5 * a
```

Note that the `/=` scales the matrix by the reciprocal of the given scalar factor.

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 4

Gridded Fields

This section describes how gridded fields are implemented in ARTS and how they are used. Gridded fields consist of a data object like a Vector, Matrix, or Tensor and a grid for each dimension of its data. For example, a GriddedField1 consists of one grid and a Vector, whereas a GriddedField3 contains three grids and a Tensor3. Grids can be either numeric, like a pressure grid, or strings, like channel names.

4.1 Implementation files

The GriddedField1, GriddedField2, GriddedField3, GriddedField4 classes and their common base class `GriddedField` described below reside in the files:

- `gridded_fields.h`
- `gridded_fields.cc`

4.2 Design

4.2.1 The abstract base class `GriddedField`

The abstract base class `GriddedField` implements the properties all gridded fields have in common. These are mostly the methods to create, set, and access the grids. A `GriddedField` is never instantiated directly.

4.2.2 Inheritance

The `GriddedFieldX` classes use indirect inheritance to combine a data object with the grids, see Figure 4.1.

History

2010-09-28 Oliver Lemke: Updated for implementation changes.
2010-04-12 Created and written by Oliver Lemke.

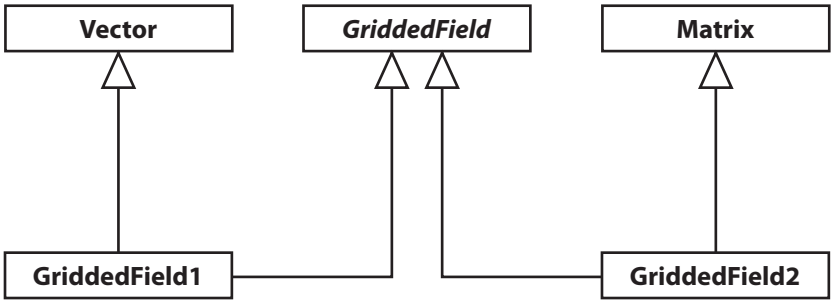

Figure 4.1: UML diagram of gridded field inheritance.

4.3 Constructing Gridded Fields

4.3.1 Creation

Each `GriddedFieldX` offers two constructors. One default constructor that creates an unnamed gridded field and a second constructor that takes a string with the name of the gridded field as an argument.

```
GriddedField1 gfone("I'm a GriddedField1");
GriddedField2 gftwo;

gftwo.set_name ("I'm a GriddedField2");
```

4.3.2 Initializing the grids

Once a gridded field has been created, we can start setting up the grids. There are two different types of grids, a numeric grid and a string grid. In the following example we set up two gridded fields: A GriddedField1 with a numeric grid and a GriddedField2 with a numeric grid for the rows and a string grid for the columns. Each grid can be assigned a name to describe its contents or unit.

```
Vector gfonegrid(1,5,1);         // gfonegrid = [1,2,3,4,5]
gfone.set_grid(0, gfonegrid);    // Set grid for the vector elements.

Vector gftwogrid0(1,5,1);        // gftwogrid0 = [1,2,3,4,5]
ArrayOfString gftwogrid1{"Chan1", "Chan2", "Chan3"};

gftwo.set_grid(0, gftwogrid0);   // Set grid for the matrix rows.
gftwo.set_grid(1, gftwogrid1);   // Set grid for the matrix columns.

gfone.set_grid_name (0, "Pressure");

gftwo.set_grid_name (0, "Pressure");
gftwo.set_grid_name (1, "Channel");
```

4.3.3 Initializing the data

The data of a `GriddedFieldX` can be accessed through its `data` member. For a GriddedField1 `data` is a Vector, for a GriddedField2 a Matrix, for a GriddedField3 a Tensor3,

and so on.

The following code shows how to fill the gridded fields from the previous example with data:

```
Vector avector(1,4,0.5);     // avector = [1,1.5,2,2.5]

gfone.data = avector;

Matrix amatrix(5,3,4.);      // amatrix = [[4,4,4],[4,4,4],...]

gftwo.data = amatrix;
```

4.3.4 Consistency check

After initializing or changing either the grids or the data, it can happen that the size of the grids does not match the size of the data anymore. Each gridded field provides a convenience function which can be called to perform a consistency check.

```
if (!gfone.checksize())
  cout << gfone.get_name()
       << ": Sizes of grid and data don't match" << endl;

// This should fail!
if (!gftwo.checksize())
  cout << gftwo.get_name()
       << ": Sizes of grids and data don't match" << endl;
```

The complete source code of the examples from this chapter can be found in `src/test_gridded_fields.cc`.

Chapter 5

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 multilinear 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.

5.1 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.

5.2 Green and blue interpolation

There are two different types of interpolation in ARTS:

History	
100204	Added documentation of grid checking functions by Stefan Buehler.
020528	Created by Stefan Buehler.

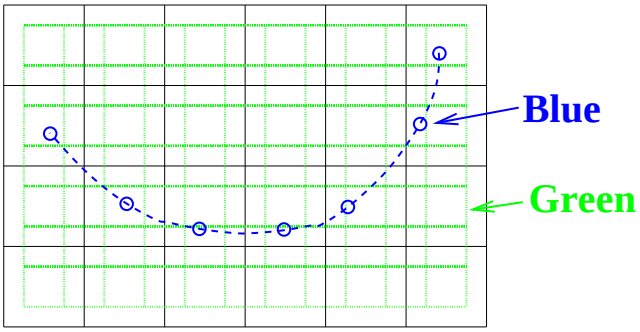

Figure 5.1: The two different types of interpolation. Green (dotted): Interpolation to a new grid, output has same dimension as input, in this case 2D. Blue (dashed): Interpolation to a sequence of points, output is always 1D.

Green Interpolation: Interpolation of a gridded field to a new grid.

Blue Interpolation: Interpolation of a gridded field to a sequence of positions.

Figure 5.1 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.

5.3 Grid checking functions

Before you do an interpolation, you should check that the new grid is inside the old grid. (Or only slightly outside.) You can use the convenience function `chk_interpolation_grids` for this purpose, which resides in file `check_input.cc`. The function has the following parameters:

```
const String&     which_interpolation   A string describing the
                                        interpolation for which
                                        the grids are intended.
ConstVectorView   old_grid              The original grid.
ConstVectorView   new_grid              The new grid.
const Numeric&    extpolfac             The extrapolation fraction.
                                        See gridpos function for
                                        details. Has a default
                                        value, which is consistent
                                        with gridpos.
```

There is also a special version for the case that the new grid is just a scalar. What the function does is check if old and new grid for an interpolation are ok. If not, it throws a detailed runtime error message.

The parameter `extpolfac` determines how much extrapolation is tolerated. Its default value is 0.5, which means that we allow extrapolation as far out as half the spacing of the last two grid points on that edge of the grid.

The `chk_interpolation_grids` function is quite thorough. It checks not only the grid range, but also the proper sorting, whether there are duplicate values, etc.. It is not

completely cheap computationally. Its intended use is at the beginning of workspace methods, when you check the input variables and issue runtime errors if there are any problems. The runtime error thrown also explains in quite a lot of detail what is actually wrong with the grids.

5.4 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`).

5.5 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
              const Numeric&  extpolfac=0.5 );
```

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!
- If you want to use the returned gp object for something else than interpolation, you should know that gridpos guarantees the following:
  For the ascending old grid case:

  ```
  old_grid[tgp.idx]<=tng || tgp.idx==0
  ```

  And for the descending old grid case:

  ```
  old_grid[tgp.idx]>=tng || tgp.idx==0
  ```

- Finally, note that parameter `extpolfac` plays the same role as explained above in Section 5.3.

5.6 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. Taking the corner points in the order indicated in Figure 5.2, the interpolated value is given by:

$$\begin{aligned} y(t,u) &= (1-t)*(1-u)*y_1 \\ &\quad + t*(1-u)*y_2 \\ &\quad + (1-t)*u*y_3 \\ &\quad + t*u*y_4 \\ &= w_1*y_1 + w_2*y_2 + w_3*y_3 + w_4*y_4 \end{aligned} \tag{5.1}$$

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).

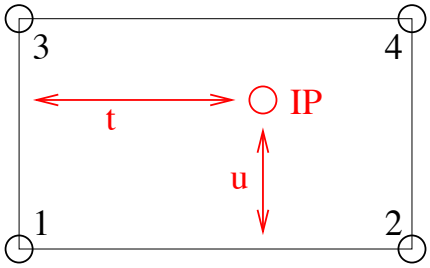

Figure 5.2: The grid square for 2D interpolation. The numbers 1...4 mark the corner points, IP is the interpolation point, t and u are the fractional distances in the two dimensions.

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).

5.7 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.

5.7.1 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.

5.7.2 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).

5.8 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 5.1). The `interp` family of functions performs this step.

5.8.1 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);
```

5.8.2 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);
```

5.9 Examples

5.9.1 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
```

5.9.2 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
```

5.10 Higher order interpolation

Everything that was written so far in this chapter referred to linear interpolation, which uses two neighboring data points in the 1D case. But ARTS also has a framework for higher order polynomial interpolation. It is defined in the the file

- `matpack/interp.h`

5.10.1 Weights

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.

As a special case, interpolation order $O = 0$ is also implemented, which means 'nearest neighbor interpolation'. In other words, the value at the closest neighboring point is chosen, so there is no real interpolation at all. This case is particularly useful if you have a field that may be interpolated in several dimensions, but you do not really want to do all dimensions all the time. With $O = 0$ interpolation and a grid that matches the original grid, interpolation can be effectively 'turned off' for that dimension.

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 `my_interp::Lagrange<>`, instead of `GridPos`. A `my_interp::Lagrange<>` object contains the grids first index, interpolation weights for all interpolation points, and on demand the linear derivative of the interpolation at the grid position. For each point in the new grid, there is 1 index, $O + 1$ weights, and 0 or $O + 1$ weight derivatives.

The `my_interp::Lagrange<>` type is a template and requires instantiation upon use of several compile-time parameters. The template signature is:

```
template <
    Index PolyOrder=-1,
    bool do_derivs=false,
    GridType type=GridType::Standard,
    template <cycle_limit lim> class Limit=no_cycle>
  requires(test_cyclic_limit<Limit>())
struct Lagrange;
```

The `PolyOrder Index` informs the type about its interpolation order. If it is negative, the object's polynomial order is determined at runtime. If it is positive, the value of the polynomial order has been determined at compile time. The difference between runtime and compile time objects is that you tend to to get orders of magnitude faster execution times if the value is known at compile time.

The `do_derivs bool` tells the type to also compute the derivatives of the weights. If this is false, fewer calculations are performed but you cannot compute the derivatives. In general, computing the derivatives add an overhead of in worst case 2, as there's often quite a lot less work to do to compute the derivatives.

The shortcodetype `GridType` selects the grid transformation. `GridType` is described more below for options, but there are two special grid types that are important to distinguish: cyclic and non-cyclic grid types. If the type is inherently cyclic, special care is taken to cycle the indices and weights so that you can interpolate over the "borders" of the input vector grid.

The `template <cycle_limit lim> class Limit` template class over the `lim cycle_limit` determines the cyclicity of the grid. It has to be `my_interp::no_cycle` for all non-cyclic grids. The template class itself is very simple. It needs to be possible to instantiate the class with `cycle_limit::lower` and `cycle_limit::upper` such that the class has a `static constexpr Numeric` member called `bound`. If the class is instantiated with the `cycle_limit::lower`, the value of `bound` must be strictly lower than the value of the class as instantiated by `cycle_limit::upper`. Three examples of cyclic bounds are provided as `my_interp::cycle_m180_p180`, `my_interp::cycle_0_p360`, and `my_interp::cycle_0_p2pi`, which respectively represents the cyclic bounds of $[-180,\ 180)$, $[0,\ 360)$, and $[0,\ 2\pi)$.

In contrast to `GridPos`, `my_interp::Lagrange<>` stores weights `lx` rather than fractional distances `fd`. For the linear case:

```
lx[0] = fd[1]
lx[1] = fd[0]
```

So the two concepts are almost the same. Because the `lx` are associated with each interpolation point, they work also for higher interpolation order, whereas the concept of fractional distance does not.

The weights along any dimension is calculated according to

$$l_j(x) = \prod_{\substack{0 \le m \le O \\ m \ne j}} \frac{u(f(x) - f(x_m))}{u(f(x_j) - f(x_m))} \tag{5.2}$$

where f is a grid scaling function and u is a combination of sign-reversal and cyclic minima. The f can be a logarithm, reverse cosine, circular constraints, or, most commonly, just the input. The provided options are part of the `my_interp::GridType` enum class and are:

`Standard` $f(t) = t$. $u(t) = t$.

`Cyclic` $f(t) = t + n(t_1 - t_0)$, where $c_0 \le t + n(c_1 - c_0) < c_1$, with n as an integer and $[c_0, c_1)$ as the cyclic limits so that $g(c_0) \equiv g(c_0 + m[c_1 - c_0])$ holds true for a valid function $g(t)$ and any integer m. $u(t) = t + X$. X is found as whichever has the absolute minimum of $t + c_1 - c_0$, t, or $t + c_0 - c_1$.

`Log` $f(t) = \ln(t)$, where $t > 0$. $u(t) = t$.

`Log10` $f(t) = \log_{10}(t)$, where $t > 0$. $u(t) = t$.

`Log2` $f(t) = \log_2(t)$, where $t > 0$. $u(t) = t$.

`SinDeg` $f(t) = \sin\left(\frac{\pi}{180}t\right)$, where $-90 \le t \le 90$. $u(t) = t$.

`SinRad` $f(t) = \sin(t)$, where $-\pi/2 \le t \le \pi/2$. $u(t) = t$.

`CosDeg` $f(t) = \cos\left(\frac{\pi}{180}[180 - t]\right)$, where $0 \le t \le 180$. $u(t) = -t$.

`CosRad` $f(t) = \cos(\pi - t)$, where $0 \le t \le \pi$. $u(t) = -t$.

The derivatives are computed as

$$\frac{\partial l_j(x)}{\partial x} = \sum_{i=0}^{O} \begin{cases} \frac{l_j(x)}{x - x_i} & \text{if} \quad x \ne x_i \\ \frac{1}{x_j - x_i} \prod_{m=0}^{O} \begin{cases} \frac{x - x_m}{x_j - x_m} & \text{if} \quad m \ne i,\ j \\ 1 & \text{if} \quad m \in i,\ j \end{cases} & \text{if} \quad x \equiv x_i \end{cases} \quad i \ne j. \tag{5.3}$$

Note that the upper branch speedup is only available for `Standard` and for `Cyclic` code. Other cases must use the lower branch to get linear derivatives.

Instead of `gridpos`, you have to use the constructor `my_interp::Lagrange<>` for higher order interpolation with a single interpolation point, and `my_interp::lagrange_interpolation_list<my_interp::Lagrange<>>` for multiple outputs. The constructor requires a start-position guess, the value at which to interpolate towards, and the original grid as inputs. In the version of `my_interp::Lagrange<>` that has its polynomial order determined at runtime, and addition number representing this polynomial order has to also be passed (so that the choice of runtime rather than compile time polynomial order is explicit). The multiple outputs function takes the new grid followed by the old grid as arguments. Again, the runtime polynomial order has to also explicitly be set when the runtime when calling this function. An optional but crucial final parameter can be passed to the function to determine if the extrapolation outside of a grid is acceptable. By default, the new grid is only allowed to be half a step size beyond the upper and lower edges of the old grid.

5.10.2 Interpolation

So far we have not computed any interpolation but just the weights. For the interpolation, the code using one or more (list of) `my_interp::Lagrange<>` can both mimic, but also differs in parts significantly from, the linear interpolation discussed above. Perhaps the most important difference is that there are no blue interpolation schemes. This was not used anywhere at the time of implementation, so it was deemed less useful. Instead, there are only two types of interpolation offered: full interpolation that goes from a N-dimensional tensor input to a scalar, and full re-interpolation that goes from one N-dimensional tensor and outputs another N-dimensional tensor. Note that we say "scalar" and not `Numeric`, because we can handle a much wider variety of input value type, perhaps most notable `Complex`.

The call order after you have a list of (lists of) `my_interp::Lagrange<>` is simple. Given `lag...` as this list and `in` as the input field, the call-order for scalar interpolation is

```
auto itw = interpweights(lag...);
auto out = interp(in, itw, lag...);
```

where `out` is a scalar. It is very important that the rank of `in` is the same as the count of the number of `lag...`. If you want to have the derivative instead of the interpolation along some dimension `dim`, the call is

```
auto ditw = dinterpweights<dim>(lag...);
auto dout = interp(in, itw, lag...);
```

where again `dout` is a scalar but now represents the derivative along the select dimension. The `dim` must be 0 or higher but strictly less than the rank of `in`. The two interpolation weight tensors `itw` and `ditw` will here have the rank as `in` with a shape that is the polynomial order plus one in the same order as `lag...`. It is possible to pre-allocate these sizes and call these two functions directly with `d/itw` as the first input. For re-interpolation, the call order is very similar

```
auto ritw = interpweights(lag...);
auto rout = reinterp(in, itw, lag...);
```

where `rout` is a tensor the same rank as `in`. If you want to have the derivative instead of the interpolation along some dimension `dim`, the call is

```
auto dritw = dinterpweights<dim>(lag...);
auto drout = reinterp(in, itw, lag...);
```

where again `drout` is a tensor the same rank as `in` but now represents the derivative along the select dimension. The rank of `ritw` and `dritw` is twice that of the rank of `in`. The inner half of the shape is exactly the same as in the scalar interpolation. The outer half of the shape is the same as the length of the lists that makes up the `lag...`.

Note that for convenience and for an unknown effect on the speed of the calculations, it is optional to compute the interpolation weights. You can call `interp` and `reinterp` directly, omitting the calls to `interpweights` and `dinterpweights`. We are to this date (2023-02-27) not sure what that does to execu5tion speed and cannot give any recommendation either way on how to use it. Different compilers seem to prefer different solutions, so it is better for code consistency to stick with the same approach as the `gridpos` does of demanding a call to `interpweights` and `dinterpweights` first.

5.11 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 `my_interp::Lagrange<>` or `my_interp::lagrange_interpolation_list<>` (one for each interpolation dimension)
2. `interpweights`
3. `interp` or `reinterp`

If you are interested in how the functions really work, look in file `interpolation.cc` or `matpack/interp.h`. 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. However, also be aware that sometimes reallocating might be preferred to passing views.

Chapter 6

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 **??** 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.

6.1 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.

6.2 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 :

$$\int_{x_1}^{x_2} f(x)dx = \frac{1}{2}h(f_1 + f_2) + O(h^3 f'') \tag{6.1}$$

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.

History	
220802	Created and written by Sreerekha T.R.
220103	Included mathematical description for implemented integration method(CE).

If we use eq. 6.1 $N-1$ 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 :

$$\int_{x_1}^{x_N} f(x)dx = \frac{1}{2}h\left[f_1 + 2(f_2 + f_3 + ... + f_{N-1}) + f_N\right] + O\left[\frac{(b-a)^3 f''}{N^2}\right] \quad (6.2)$$

The last term tells how much the error will be decreased by taking more number of steps.

6.3 Solid Angle Integration

In our scattering problem, we are often encountered with a double integration of functions over zenith and azimuth angles (see Chapter **??**). 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

$$S = \int_{4\pi} f(\omega)\mathrm{d}\omega \quad (6.3)$$

In spherical coordinates we can write:

$$S = \int_0^{\pi}\int_0^{2\pi} f(\theta,\phi)\sin\theta \quad \mathrm{d}\theta\mathrm{d}\phi \quad (6.4)$$

A double integration can be splitted into two single integrations:

$$S = \int_0^{\pi}\left(\int_0^{2\pi} f(\theta,\phi)\sin\theta\mathrm{d}\phi\right)\mathrm{d}\theta \quad (6.5)$$

$$= \int_0^{\pi} g(\theta)\mathrm{d}\theta \quad (6.6)$$

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.

$$g(\theta_j) = \sum_{i=1}^{m} \sin\theta_j \frac{f(\theta_j,\phi_i) + f(\theta_j,\phi_{i+1})}{2} \cdot (\phi_{i+1} - \phi_i) \quad (6.7)$$

The final step is to sum up all $g(\theta_j)$, again applying the trapezoidal method.

$$S = \sum_{j=1}^{n} \frac{g(\theta_j) + g(\theta_{j+1})}{2} \cdot (\theta_{j+1} - \theta_j) \quad (6.8)$$

If the radiation is symmetric about the azimuth we just calculate:

$$S_{sym} = 2\pi \int_0^{\pi} f(\theta) \sin(\theta) \mathrm{d}\theta \tag{6.9}$$

Unsing the trapezoidal method this can be written as:

$$S_{sym} = 2\pi \sum_{j=1}^{n} \frac{h(\theta_j) + h(\theta_{j+1})}{2} \cdot (\theta_{j+1} - \theta_j) \tag{6.10}$$

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 7

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.

7.1 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.

7.2 Linear Equation Systems

For solving a set of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{7.1}$$

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 (7.1):

- 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 7.2.1):
```
ludcmp(LU, indx, A);
```
- Call LU backsubstitution function (see Section 7.2.2):
```
lubacksub(x, LU, b, indx);
```
- Print the solution vector:
```
cout << x;
```

7.2.1 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):

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{7.2}$$

For a 3 x 3 matrix equation 7.2 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 (7.1) in the following way:

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{7.3}$$

First

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{7.4}$$

is solved for the vector $\mathbf{y}$ which can be done by forward substitution (see section 7.2.2). Then

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{7.5}$$

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.

7.2.2 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

$$x_3 = b_3/a_{33} \tag{7.6}$$

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 7.2.1. 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.

7.2.3 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 (Equation **??** in *ARTS User Guide*) 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 7.1.

$$\mathbf{K}^{-1} \cdot \mathbf{b} = \mathbf{x} \tag{7.7}$$
$$\Leftrightarrow \quad \mathbf{K} \cdot \mathbf{x} = \mathbf{b} \tag{7.8}$$

- To solve the equation system

$$\mathbf{A} \cdot \mathbf{X} = \mathbf{B} \tag{7.9}$$

  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.

7.3 Matrix Exponential Function

A very important function for solving differential equations is the matrix exponential:

$$e^{\mathbf{A}s} = \sum_{k=0}^{\infty} \frac{(\mathbf{A}s)^k}{k!} \tag{7.10}$$

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.

7.3.1 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

$$\mathbf{F} = e^{\mathbf{A}+\mathbf{E}} \tag{7.11}$$

where

$$\|\mathbf{E}\|_\infty \leq \delta \|\mathbf{A}\|. \tag{7.12}$$

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

$$\epsilon(p,q) = 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}. \tag{7.13}$$

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 $\mathbf{A}$ and assign values:
```
Matrix A(3,3);
A(1,1) = 45;
A(1,2) = 3;
...
```
- Initialize $\mathbf{F}$:
```
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;
```

Chapter 8

Include ARTS in third-party C++

It is possible to include ARTS in another C++ program using the `public_arts_interface` and linking to `autoarts.h`. This will pull in the ARTS public API defined in `agendas.cc`, `groups.cc`, `methods.cc`, `workspace.cc` as well as an automatically generated ARTS namespace to your project.

FIXME: a private Module that imports only the ARTS namespace should be added as soon as soon as C++20 becomes norm

8.1 Linking the public interface

To link the public interface, you need to `add_subdirectory(arts)` anywhere in your CMake project, where the `arts` directory should contain a current version of ARTS.

An example (in your projects `CMakeLists.txt`):

```
#######################################################
# ARTS Custom Executable
add_executable(arts_interface interface.cpp)
target_link_libraries(arts_interface PUBLIC
                      public_arts_interface)
#######################################################
```

At this point, your `interface.cpp` must have `#include <autoarts.h>` as one of its headers and you are good to go.

8.2 Using the C++ namespace interface

The ARTS namespace contains all the interfaces you will need to perform all operations supported by ARTS. The namespace defines only a single top level function call, the `init(...)` function and a single top-level type, the ARTS Workspace. The function is used to generate the ARTS Workspace upon which all your function calls are made. The ARTS namespace has several sub-namespaces for various purposes. These are in short:

Group Defines the ARTS public type-system.

History

2020-10-06 Created and written by Richard Larsson.

Internal Defines the ARTS internal type-system.

Var Defines the ARTS interface type-system, create variables for the ARTS interface, and access automatic variables that are defined in the ARTS Workspace from the start.

Method Defines the ARTS methods. The ARTS methods can only be called using their generic inputs and outputs.

AgendaMethod Defines the ARTS methods that can be used by agendas. They return an internal type and are best used directly, with no manual modification.

AgendaDefine Defines methods to set the different ARTS agendas. The agendas are checked and ready to be used after these methods are called.

AgendaExecute Executes an agenda.

If you are using the public interface, you need not be concerned with the types in the sub-namespaces Group and Internal. **FIXME: Otherwise, these define the basic types you need to use ARTS easily.** These sub-namespaces are used the same as any other C++ type-system. The other namespaces are domain specific.

An interface will generally start with a call to the initialize function. This could look like:

```
auto ws = init();
```

After this the order and set of commands that are placed is up to the user. For sake of ease, `ws` will be used below to indicate a defined Workspace. Also, the access to each sub-namespace will be written as if operating in the ARTS namespace itself.

8.2.1 Var

The Var namespace, short for Variable namespace, have three purposes

1. Type the Method and Agenda interface
2. Access common Workspace variables
3. Create new variables on the Workspace

The types that are defined correspond to the types in Group. The purpose of these types is to pass input to the functions of Method and AgendaMethod. The main way to generate instances of these variables is by their corresponding `*Create` function or by simply accessing them via their common Workspace names. The only constructor that is recommended to use is the construction from the corresponding Group, as this can simplify the access to several methods. The other two constructors risk accessing out-of-bound memory, or to deference a null-pointer.

It is highly recommended to not discard created variables, as they will still occupy memory in the Workspace until the end of the program and they become impossible to access once discarded.

Examples:

```
// Define an index that is not in the workspace
Var::Index x{Group::Index(1)};

// Access an index that is in the workspace
Var::Index y = Var::stokes_dim(ws);

// Create a new index on the workspace
Var::Index z{Var::IndexCreate(ws, Group::Index{1},
                              "new_index_name")};
```

Note that `x` will not work as an input to AgendaMethod functions but it will work as an input to Method functions. It does not append to the Workspace. The other two will work both as inputs to AgendaMethod and to Method functions.

8.2.2 Method

The Method namespace contains all but the Agenda manipulating methods defined in `methods.cc`. These can all be called using the generic input and outputs. The inputs and outputs are not guaranteed to be the same as in the ARTS methods however, because C++ requires inputs with default values be placed last in the calling order. Note that all standard inputs and outputs taken from the Workspace must be set on the Workspace itself and cannot be passed as inputs. This creates a few idiosyncrasies compared to how ARTS is used in python or in a normal controlfile.

Examples:

```
// Call yCalc (no GIN/GOUT)
Method::yCalc(ws);

// Call Touch on the wind field (Pure GOUT)
Method::Touch(ws, Var::wind_u_field(ws));
Method::Touch(ws, Var::wind_v_field(ws));
Method::Touch(ws, Var::wind_w_field(ws));

// Set p_grid by VectorNLogSpace
Var::nelem(ws) = 51;
Method::VectorNLogSpace(ws, Var::p_grid(ws), 1e+05, 1e-4);

// Save x, y, z from the Var example
Var::output_file_format(ws) = Group::String{"ascii"};
Method::WriteXML(ws, x);
Method::WriteXML(ws, y, Group::String{"extra.xml"});
Method::WriteXML(ws, z);
```

All methods requires a Workspace (`ws`) to work. The first case of the examples calls a function with neither generic input nor generic output — it cannot take anything other than the Workspace. The second triplet case of the examples calls Touch on all wind-field variables. They will have been default-initialized after this process. The third example shows the idiosyncrasies to other methods of using ARTS. The Workspace variable `nelem` has been used instead of a generic input index to define `VectorNLogSpace` — thus `nelem`

must be set manually by the user of the interface. The last examples uses the fact that many of `WriteXML`'s inputs are default defined. Since the default value of filename is empty and since the interface then infers it has the variable name input, the first call to `WriteXML` will generate a file called `arts.in.xml`, the second call will generate a file called `extra.xml`, and the last call will generate a file called `arts.new_index_name.xml`. The first part of the name can be changed by calling `init()` with the corresponding arguments.

As a last note. Several inputs above automatically generates inputs from standard C++ types, such as `VectorNLogSpace` generating two `Var::Numeric` from two doubles. This is a convenient way to use the methods but the user should be aware that these methods will end up deleting variables in the end, so some care has to be taken when the scope of such automatic variables is long.

8.2.3 AgendaMethod, AgendaDefine, and AgendaExecute

The Agenda namespaces deal with setting and defining Workspace Agendas. It is only possible to set Workspace Agendas that have been defined as part of the Workspace at compilation time. The AgendaMethod namespace contains the same functions as the Method namespace but returns an Internal::MRecord. The user of this interface is expected to never manually construct an MRecord. Instead, this type is meant to only be used when Agendas are defined in the AgendaDefine namespace. The AgendaDefine namespace defines a variadic function per Agenda in the common Workspace and expects a list of MRecord to set this Agenda's methods. Finally, AgendaExecute exists to execute a single Agenda. Normally, this is not preferred since Agendas should generally just be used inside methods, but the option still exists.

Examples:

```
// Define the basic iy_main_agenda emission agenda
AgendaDefine::iy_main_agenda(ws,
  AgendaMethod::ppathCalc(ws),
  AgendaMethod::iyEmissionStandard(ws));

// Define an empty geo positioning agenda
AgendaDefine::geo_pos_agenda(ws,
  AgendaMethod::Ignore(ws, Var::ppath(ws)),
  AgendaMethod::VectorSet(ws, Var::geo_pos(ws),
    Var::VectorCreate(ws, {}, "Default")));
```

The first example simply sets its agenda by using two functions that complete the inputs/outputs expected of the agenda. The second example does not need or want to use `Var::ppath(ws)` so it ignores it. It also has the need of a default empty `Var::Vector`. This variable has to first be created onto the Workspace before it can be used as an input. Had a pure vector been used instead of the create-function, a runtime error would occur. The create function is only invoked once, since the internal workings of the Agenda just need to have defined the variable.

Lastly, each AgendaMethod function that has a default generic input will create a static const Workspace variable of this type upon first call to the method. This means that calling such a method will incur a memory cost that lasts until the end of the program.

Chapter 9

GUI — simple plotting

ARTS provides a simple debug plotting tool for developers to see their results in-place. To build and use the plotting tool you need to activate `-DENABLE_GUI=1` in your cmake settings. The plotting debug tool is now available by using `#include <gui/plot.h>` in your desired file. The plotting routines that are available are quite simple. You get these interfaces

```
plot(const Vector& y)  // Plots y evenly spaced
plot(const Vector& x, const Vector& y)
plot(...)  // Plots any number of pairs of X-Y
```

These are all available via the `ARTSGUI` namespace. Figure 9.1 shows a fullscreen example produced by the templated version of plot on Ubuntu.

The features of the plotting tool is limited but intuitive enough to access and view the data.

There are three components to the plotting tool. The plot frame is where your data is displayed. The plot axis shows the scales of the data. The menu bar offers some options to manipulate the window. Each of these have limited operations listed below.

- Menu bar
  - `File` — access meta information about the window and data
    - `Fullscreen` (shortcut F11) takes the plotting window fullscreen. The exit fullscreen mode, use escape, F11, quit the application or press the fullscreen menu item again.
    - `Export Data` (shortcut Ctrl+S) saves the Y-axis data to an XML-file. In the templated code, the new file will be an ArrayOfVector, in the non-templated code it will be a simple Vector.
    - `Quit` (shortcut Ctrl+X) quits the application.
- Plot axis
  - Double left-click: Resets the clicked axis to the maximum possible value. Note that for values very small or very large, this might fail and you need to manually fix the axis.

History

201203 First version Richard Larsson

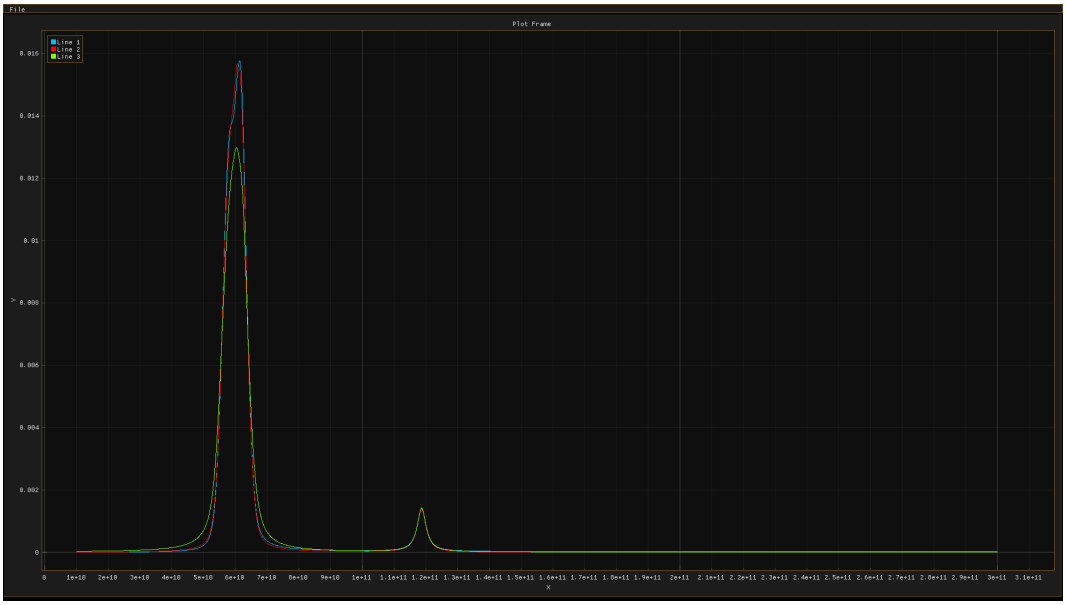

Figure 9.1: Example plot. Here we show the absorption versus frequency for some O_2 absorption models.

  - Double right-click: Opens the axis dropdown-menu with the following options:
    * Min: Select the minimum value by writing the number in the text box, or lock the axis by ticking the checkbox.
    * Max: Select the maximum value by writing the number in the text box, or lock the axis by ticking the checkbox.
    * Invert: Check the box to switch the order of high-and-low along the respective axis.
    * Log Scale: Check the box to show the axis in logarithmic scale.
    * Time (X-axis only): Check the box to interpret the X axis as Unix time and display the relevant as dates.
    * Grid Lines: Uncheck the box to remove grid lines in the plot frame for the axis.
    * Tick Marks: Uncheck the box to remove tick marks in the plot frame for the axis.
    * Labels: Uncheck the box to remove the labels from the axis.
  - Left-click and drag: Moves the axis in the direction dragged.
  - Vertical scroll: Extends or contracts the axis range around the value where the scroll began.

- Plot frame
  - Double left-click: Scales both X-and-Y to a best-fit scenario (still with the caveat that very small and very large numbers might not work properly)
  - Double right-click: Opens a dropdown-menu with access to the plot axis dropdown-menues and some settings
  - Scroll: Extend or contract both axis around the point where the scoll began

- Left-click and drag: Moves the axis around
- Right-click and drag: Selects a region and zooms in when the right-click is let go. If Shift is held, the zoom window is modified to include the entire Y-axis. If Alt held, the zoom window is modified to include the entire X-axis.
- Legends: In the upper left corner of the plot frame are the legends of the plot. If hovered, the hovered line will be enlarged inside the plot frame. If the colored check box is pressed, the line will be removed from the plot frame.

The debug GUI is based on the `Dear ImGui` GUI framework, with `ImPlot` drawing the plots and uses `imgui-filebrowser` to deal with the filesystem.

Chapter 10

The Python interface

This section is devoted to describing various techniques and code used to enhance the Python-ARTS interface: `pyarts`. It will not be exhaustive but will go over the basic ideas of how to add and modify classes and functions.

The basic section will explain where the files are located in an ARTS directory and what the general build steps involved. The classes section described how new classes are added and modified. The workspace and agenda classes are special and so they get their own section. The functions section will be short and go over how functions are added.

10.1 Basics

A `pyarts` installation consists of two parts: the pure Python code available in the main directory `python/pyarts/` folder and the C++ interface. The C++ interface relies on the `pybind11` Python bindings package available via `https://github.com/pybind/pybind11`. We keep a local copy of the latest release of `pybind11` around in `3rdparty/pybind11/`. `pybind11` is a pure headers only library and these headers can be found at `3rdparty/pybind11/includes/pybind11`. The `CMake` settings ensure that these can be included by any file compiled as part of the `pyarts_cpp` target using `#include <pybind11/*.h>`. The ARTS side of C++ linkage is available in the `src/python_interface` folder. There are two parts: the auto-generated files that are only available after compiling the target, and the other static files.

The `py_module.cpp` file links everything together. It is responsible for passing a module reference object along to all other functions. To keep things simple, we avoid creating a `py_module.h` file and simply declare that the functions from the other files are available to the linker down-the-line.

The steps to add a new file with new functionality are as follows (for `py_my_file.cpp` as the file and `my_new_functionality` as function):

1. Create the `py_my_file.cpp` file
2. Add `py_my_file.cpp` to the `pyarts_cpp` target in the `src/python_interface/CMakeLists.txt`.
3. Fill the file with the relevant includes (see below).
4. Within `namespace Python` setup `void my_new_functionality(py::module_& m) {*}`.

5. Replace `*` with your new functionality (and run the automatic formatter)
6. Add `void my_new_functionality(py::module_& m);` directly within the `Python` namespace of `py_module.cpp`.
7. Add `my_new_functionality(m);` within the `PYBIND11_MODULE(pyarts_cpp, m)` function definition of `py_module.cpp`.
8. Compile. The new functionality is available via `pyarts.pyarts_cpp.*`.
9. Add a continuous integration test that tests the new functionality in a limited fashion.

10.1.1 Includes

To keep a consistent experience across the code, it is recommended to put either `#include "python_interface.h"` or `#include <py_auto_interface.h>` at the top of the files. This ensures that you have access to almost all of `pybind11` and `artscore`. If you need any of the headers that are not included from `artscore`, the `src/` folder is also linked to the `pyarts_cpp` target so you can include it without using relative paths.

Note that the documentation of the Python functions and classes that are generated depends on the order of calls within `py_module.cpp`. If your class depends on `matpack` and on gridded fields, place the call after these. If in turn the Jacobian calculations depends on your type, place the call above it. This only applies to the call to your functionality (not the declaration above the module creation macro).

10.1.2 Limitations

There are many limitations. Most of these are weeded out by keeping a constantly up-to-date set of tests available to be run as part of the continuous integration. Any feature not tested that way will be considered unwanted as part of the Python interface. Add new tests as you add new functionality! This is very important as `pybind11` links directly to Python, meaning that new features might work on your computer but not on others. Even worse, changes in the future might break your feature.

One limitation we cannot work around is that there are no references to `int` and `float` in Python. The ARTS `Index` and `Numeric` map directly to these Python types. As we need references to these variables, wrappers known as `Index_` and `Numeric_` have been created. If you need references, ensure that you are returning these types instead of the pure types. Note that there exist a helper function `as_ref` that can deal with this problem for you. Only use this if you know that you have a reference that will persist to a base-type.

A limitation that we can work around is that Python and `pybind11` does not have the concept of constants and references in general. If the output of a function is a reference, we need to mark that that variable is not allowed to be converted. More about this below.

Lastly, we often compile ARTS without debug information, turning off several sanity checks inside the classes. These might cause segmentation fault if encountered in Python. Please ensure that you do proper input/output error handling manually when mapping to these functionalities.

10.2 Classes, structs, and enums

These are the core data holders in ARTS and the Python interface aims to wrap them in the way that their use in Python mostly mimics their use in ARTS. Of course, you might sometimes need to modify ARTS classes to get the desired Python behaviour.

It is important that all classes that can be returned or used as arguments in the Python interface are defined as such. The only exceptions to this rule are inputs that can be automatically converted from standard types to Python types and back by the `pybind11` automatic conversion functions.

10.2.1 Adding new class or struct

A new class and struct is registered as

C++
```
auto c = py::class_<NewClassOrStruct>(m, "NameInPython");
```

This new class will be registered as the type `pyarts.pyarts_cpp.NameInPython`. It is also mapped via `pyarts.classes.NameInPython`. Here `NewClassOrStruct` is a class or struct that the compiler is aware of from C++, `m` is the common module name (see above), and `c` variable can accessed to modify the functionality of the class as seen in Python. Most of these modifications returns `*this` as a reference. It is thus often more convenient to write

C++
```
py::class_<NewClassOrStruct>(m, "NameInPython")
    .def(...)
    .def_property(...)
    .def_readwrite(...)
    ...
```

Below the use of these `def*` will be discussed.

The main reason to use the former rather than the latter technique to define classes and structs is if you need the name registered in the module early on. As mentioned above, this changes how the Python documentation looks, so it is important to register names in a timely fashion.

10.2.2 Adding new enum

There are two types of enumeration in C++. Enum classes and C-style enums. It is strongly recommended to make use of the ARTS `ENUMCLASS` macro when creating new enum classes in the future. These behave very much like normal classes, providing both stream operators and conversions from and to string views.

If you want to add an `enum class` created by this macro, the following should help (from `py_jac.cpp`):

C++
```
py::class_<Jacobian::Atm>(m, "JacobianAtm")
    .def(py::init([]() { return new Jacobian::Atm{}; }))
    .def(py::init([](const std::string& c) {
          return Jacobian::toAtmOrThrow(c);
```

```
        }), py::arg("str").none(false))
    .PythonInterfaceCopyValue(Jacobian::Atm)
    .PythonInterfaceBasicRepresentation(Jacobian::Atm)
    .def(py::pickle(
        [](const Jacobian::Atm& t) {
          return py::make_tuple(std::string(Jacobian::toString(t)));
        },
        [](const py::tuple& t) {
          ARTS_USER_ERROR_IF(t.size() != 1, "Invalid state!")
          return new Jacobian::Atm{
              Jacobian::toAtm(t[0].cast<std::string>())};
        }));
py::implicitly_convertible<std::string, Jacobian::Atm>();
```

This will allow you to read and write to the `enum` using Python `str`.

If you have an `enum class` that does not derive from `ENUMCLASS` or a pure C-style `enum`, the following example should help:

C++

```
py::enum_<PType>(m, "PType")
    .value("PTYPE_GENERAL", PType::PTYPE_GENERAL)
    .value("PTYPE_AZIMUTH_RND", PType::PTYPE_AZIMUTH_RND)
    .value("PTYPE_TOTAL_RND", PType::PTYPE_TOTAL_RND)
    .PythonInterfaceCopyValue(PType)
    .def(py::pickle(
        [](const PType& self) {
          return py::make_tuple(static_cast<Index>(self));
        },
        [](const py::tuple& t) {
          ARTS_USER_ERROR_IF(t.size() != 1, "Invalid state!")

          return static_cast<PType>(t[0].cast<Index>());
        }));
```

This allows values to be accessed using, e.g., `pyarts.pyarts_cpp.PTYPE_GENERAL`. The `.value` register a name of a value of the enum.

10.2.3 Conversion from Python to ARTS types

As said about the `"JacobianAtm"` example above, it is possible and quite easy to initialize ARTS types from Python types. This works by making use of the `def(py::init(*))` method. The `*` should be replaced by a lambda that takes any number of arguments and returns a raw pointer or a value of the type that is being initialized. (Note that it does not work to use smart pointers here as the Python interface needs to take ownership.)

The `def(py::init([](const std::string& c) {*})` call from the example above, for instance, makes it possible to call

Python

```
import pyarts.pyarts_cpp as cxx
x = cxx.JacobianAtm("Temperature")
```

in Python. A `std::string` is not a Python type, but `pybind11` provides several automatic conversions between Python and some standard C++ types. These conversions generally depend on the right files being included. Likewise, we can define our own automatic conversions using `py::implicitly_convertible<From, To>();`. Since we registered the conversion from a C++ string to a `Jacobian::Atm` above, almost any functionality that takes this type (e.g., a call to a function) also accepts a Python string. The one exception here is if the function argument has been explicitly marked to not allow any conversions.

The lambda function initializing a class can take any number of variables. These can also have default values. More about that later.

Feel free to add as many implicit conversions to a class as you see fit, but remember to always have the accompanying initialization function defined when you do so. The more implicit conversions we can guarantee to work, the easier it will be to use ARTS from Python.

10.2.4 Modifying an existing class

We can add pretty much any functionality to a Python class. This section will list the ones we use. For more details on how function arguments work, see the relevant section further down. This section assumes you have `c` defined as in subsection 10.2.1. For short we will say that the C++ class `c` represents is `MyClass`.

Note that this subsection will only deal with non-static modifications. These must always have the class itself as the first argument, just as Python. All of these options have static versions as well, where the only difference is that there is no first property. The exception here is the `py::init` function, which does not require an object already but must return it.

Several of these methods can be modified for documentation and other purposes. These are detailed in the function subsection below. Also in this subsection, there will be more advanced variable usage demonstrated.

Adding a Property

Properties are added as

C++
```
c.def_property("property_name", reading, writing);
```

This property is now available as a normal Python property in all instances of `c` in Python. Here `reading` and `writing` are instances of `py::cpp_function`. This class can be constructed from a simple lambda function or be allowed more arguments. Note that if `reading` is constructed from a lambda function, it will return a copy of the object. It might be better to allow it to return a reference to the object. This can be done by changing the return value policy as

C++
```
c.def_property("property_name", py::cpp_function(reading,
    py::return_value_policy::reference_internal), writing);
```

Here, `reading` is again just a lambda function that returns a reference. The `writing` function should not return anything.

Syntax for reading and returning a value:

C++
```
// lambda:
reading := [](MyClass& x) {return x.my_property_value();}

// pointer:
reading := &MyClass::my_property_value
```

Syntax for reading and returning a reference to a value:

C++
```
// lambda:
reading := py::cpp_function([](MyClass& x) ->
  std::remove_cv_t<std::add_lvalue_reference_t<
          decltype(x.my_property_value())>> {
    return x.my_property_value();
  }, py::return_value_policy::reference_internal)

// pointer (if x.my_property_value() is a non-constant reference):
reading := py::cpp_function(&MyClass::my_property_value,
    py::return_value_policy::reference_internal)
```

Note that the long term to the right of the right arrow (`->`) is just C++ type-trait deduction to enforce that the return type has to be a non-constant reference. This is mostly a compile time test in case at any point the implementation of `MyClass::my_property_value` changes. It does not need to be there. Unless you have an identically named `x.my_property_value()` that also returns a constant reference.

The writing lambda should be defined as

C++
```
// lambda (if x.my_property_value() is a reference):
writing := [](MyClass& x, decltype(x.my_property_value()) y) {
    x.set_my_property_value() = y;
}

// lambda (if x.set_my_property_value(y) sets the value):
writing := [](MyClass& x, decltype(x.my_property_value()) y) {
    x.set_my_property_value(y);
}

// pointer (if x.set_my_property_value(y) sets the value):
writing := &MyClass::set_my_property_value
```

It might be worth here to play around with the input type. Some input is better as copies with a corresponding `std::move`, others might need a constant reference instead.

Note that properties are sometimes weird workarounds around for C++ classes that provides get- and set-functions instead of exposing member variables. If you really want to have properties on your class, it might be worth the effort to modify the original ARTS class to expose its members variables in public instead. Why would you allow direct manipulation of data this way in Python but not in C++ if the setter does not perform any additional logic or checks on the set value?

Adding a modifiable value

Any publically exposed variable of a class can be read from or written to in Python using

```C++
c.def_readwrite("readwrite_name", &MyClass::value)
```

As for with properties, you can also add a return value policy if you wish the reading to not return a copy. This is done as:

```C++
c.def_readwrite("readwrite_name", &MyClass::value,
    py::return_value_policy::reference_internal)
```

Note that from Python, this will look exactly like a standard property.

Adding a function

A function is defined as

```C++
// lambda:
c.def("function_name", [](MyClass& f, ...) {
    return f.function(...);
});

// pointer:
c.def("function_name", &MyClass::function);
```

where the ellipsis indicates any number of arguments in the first example. Of course, you have to match this to the number of arguments that are actually required.

Helper macros

The file `py_macros.h` contains several helper macros. They all look like `PythonInterface*`. Their use is not required but it is recommended to make use of them if you want to reproduce functionality. This document will not go through these macros at all. It is advised to copy their usage from already defined classes.

10.2.5 Using previously declared options

It is possible to use previously defined functions by passing the class itself into C++ as a `py::object&`. For instance:

```C++
c.def("call_function_name_with_readwrite_name",\
    [](py::object& self) {
    return self.attr("function_name")(
        self.attr("readwrite_name"));
})
```

will call the function defined above with the read-write property also defined above.

10.2.6 Monkey patching

Monkey patching is when a functionality is added to a class at runtime. We need to make use of this to simplify features such as conversions to `xarray` or to make use of `numpy` or `scipy` when deemed necessary. If possible, this should be avoided as far as possible because it has several drawbacks, one of which is that it is relatively slow.

Anyways, the solution for monkey patching we have gone with in ARTS is to retain static function pointers that are overwritten when `pyarts` is imported and, crucially, reset when `pyarts` is destroyed. The latter is important to not create segmentation faults.

In C++, the following code is required as one of the functions defined for the `MyClass` interface:

C++
```
c.def("monkey_function", [](py::object& self, ...) {
    return details::MyClass::monkey_function(...);
})
```

This will call a function that must have been defined statically inside a struct called `MyClass` that's part of the `details` namespace. Outside the function that is declaring the `MyClass` interface but still within the Python namespace, this struct may be defined as:

C++
```
namespace details {
struct MyClass {
    inline static std::function<py::object(py::object&, ...)>
      monkey_function{
        [](py::object&, ...) {
            throw std::logic_error("Not implemented");
            return py::none();
        }
    };
};
}  // namespace details
```

Note that we provide the `details.h` file to help this, so in case the function takes two arguments, you can write

C++
```
namespace details {
struct MyClass {
    inline static auto monkey_function{two_args};
};
}  // namespace details
```

Now again inside the function we need to define a way to overwrite this from Python and to cleanly destroy it. To overwrite the function, we consider it best practice to hide the class from normal users by defining it as

C++
```
py::class_<details::MyClass>(m, "MyClass::details")
    .def_readwrite_static("monkey_function",
                          &details::MyClass::monkey_function);
```

This can only be overwritten in Python using the `getattr` built-in function such as

Python
```
import pyarts.pyarts_cpp as cxx
def print_two(a, b): print(a, b)
getattr(cxx, "MyClass::details").monkey_function = print_two
```

The importance to actually destroy the variable upon deleting `pyarts` also exist. We can do this as

C++
```
m.add_object("_cleanupMyClass", py::capsule([]() {
                 details::MyClass::monkey_function =
                   details::two_args;
             }));
```

The trick here is that there will exist a `"_cleanupMyClass"` on the `pyarts.pyarts_cpp` namespace. This is ugly and technically users can overwrite it. However, it is guaranteed that when a module (like `pyarts`) is destroyed, it will first destroy all variables beginning with a single underscore. Thus the first thing that will happen is that the `details::MyClass` struct will be restored to its original state and it will not have dangling pointers when the Python function `print_two` is destroyed.

Note: do not define any monkey patch Python function with a name containing a leading underscore. This might lead to segmentation faults when closing down `pyarts`.

10.3 Workspace and Agenda

The workspace and agenda are special classes in ARTS. An agenda can only exist on a workspace but the linkage to that workspace is implicit and not explicit.

Normally, there should be no problems adding new types and functions to the workspace within the scope of the current implementations. The file `gen_auto_py.cpp` will generate the interface for you.

Note that each new workspace group has to define a large set of functionality defined by the `TestClassesBasic.py` continuous integration test.

10.4 Functions

Functions can be defined on the Python module in C++. Given the module `m` as used above, this is done with

C++
```
// lambda:
m.def("module_level_function",
    [](...) {return module_level_function(...);});

// pointer:
m.def("module_level_function", &module_level_function);
```

Here again the ellipsis means any number of arguments. This is pretty much identical to how functions are defined for classes, but the first variable does not have to be a reference to `self` or `this`.

Note again that all the classes and types that this function accept must have been defined either as automatically convertible from standard Python types, or as classes or enums in the way detailed above.

The main point of this section is to go over some of the additional options that can be given to the function definition to change and modify their behaviour.

10.4.1 Keep alive

This is defined as

C++

```
// lambda:
m.def("module_level_function",
    [](...) {return module_level_function(...);},
    py::keep_alive<Nurse, Patient>());

pointer:
m.def("module_level_function", &module_level_function,
    py::keep_alive<Nurse, Patient>());
```

where the nurse and the patient are indices indicating the value which keeps another value alive. (The nurse keeps the patient alive.) The indices here are important to keep track of. An index of 0 means the function's return value. An index of 1 means the first input of the function (for classes, this means the object itself). 2, 3, 4, and so on is thus the number of the argument as passed to the function.

The main effect of keeping an object alive with the help of another is that it stops you from having dangling references. Most of the time you do not want pure functions returning references, but it is common practice for classes. For instance, the `matpack` types can all return a non-owning `numpy` array using the `value` property. This array is not a reference to any value inside the type, yet the `numpy` array needs the `matpack` type to remain alive for as long as it exists. It thus sets `py::keep_alive<0, 1>()` so that the return value of the `value` getter keeps the main object alive for as long as it exists.

10.4.2 Return value policy

This is defined as

C++

```
// lambda:
m.def("module_level_function",
    [](...) {return module_level_function(...);},
    py::return_value_policy::*);

// pointer:
m.def("module_level_function", &module_level_function,
    py::return_value_policy::*);
```

where the * is replaced by the policy. This determines how Python views the ownership of the returned object. For instance, the policy: `reference_internal` is used by all array access getters. This ensures that the returned Python object keeps the array alive and that also that the Python object is not the owner of itself. The difference with pure keep alive is small but still there.

10.4.3 Function arguments

This is defined as:

C++

```
// lambda:
m.def("module_level_function",
     [](...) {return module_level_function(...);},
     py::arg("val"),
     py::arg("foo")=1,
     py::arg_v("bar", nullptr, "None"),
     py::arg_v("foobar", 5, "5").noconvert(),
     py::arg_v("valfoo", 5.0, "4").none(false),
     py::arg_v("valbar", Vector(5),
       "JustAVector").noconvert().none(false)
);

// pointer:
m.def("module_level_function", &module_level_function,
     py::arg("val"),
     py::arg("foo")=1,
     py::arg_v("bar", nullptr, "None"),
     py::arg_v("foobar", 5, "5").noconvert(),
     py::arg_v("valfoo", 5.0, "4").none(false),
     py::arg_v("valbar", Vector(5),
       "JustAVector").noconvert().none(false)
);
```

for a function that takes 6 arguments.

- The first argument will be named `"val"`. It has have no default value. The documentation will give no default.
- The second argument will be named `"foo"`. It has the default value of 1. The documentation will give an automatic default format.
- The third argument will be named `"bar"`. It has the default value of **`None`** and the documentation text will say so.
- The fourth argument will be named `"foobar"`. It has the default value of 5 and the documentation text will say so. It cannot be implicitly converted to the type it has but must have an exactly matching type to whatever `module_level_function` takes as a fourth argument.
- The fifth argument will be names `"valfoo"`. It has the default value 5.0 but the documentation will say it is 4. It cannot be set from **`None`**.

- The sixth argument will be names `"valbar"`. It has the default value of `Vector(5)` but the documentation will say it is `JustAVector`. It cannot be **`None`**. It is also not possible to set the sixth argument from any other type than whatever `module_level_function` takes as a sixth argument.

Variant

The C++ interface allows using `std::variant` to represent that a function can have more than one type as input. You are able to do this:

C++
```
m.def("varfun", [](std::variant<Numeric_,
                                Index_,
                                Vector> x) {...} );
```

or this

C++
```
m.def("varfun", [](Numeric_ x) {...} );
m.def("varfun", [](Index_   x) {...} );
m.def("varfun", [](Vector   x) {...} );
```

to state this in C++. The main difference is that the latter will generate a lot more documentation.

Note that there is a small problem with the functions above. This is what will happen in both cases

Python
```
import pyarts.pyarts_cpp as cxx
cxx.varfun(1)   # calls m.def("varfun", [](Numeric_ x) ...
cxx.varfun(cxx.Index(1))   # calls m.def("varfun", [](Index_   x) ...
```

The reason is that the constant 1 is a Python `int`. Since we allow the numerical class of ARTS to be created from integers, and since it is declared before the integer version of the function, the numerical call to the function has precedence. To call the integer version of the function, you have to place it above or match its type exactly. The order of resolution for function calls with overloads are:

1. Check if an exact overload for this type exists and use it.
2. Try to convert the input to one of the overloads starting from the first declared overload and working through the rest until any such conversion works. Call that function.

Optional

Values can be made optional using

C++
```
m.def("optfun", [](std::optional<Numeric> x) {...} );
```

This allows calling the function with **`None`** as input, which will simply leave `x` valueless.

Documentation

To add documentation to a function do this

C++

```
// lambda:
m.def("module_level_function",
    [](...) {return module_level_function(...);},
    py::doc("..."));

// pointer:
m.def("module_level_function", &module_level_function,
    py::doc("..."));
```

The function will now have the ellipsis as its Python documentation. Please follow best practices for Python documentation while writing these strings. Also note that it is very convenient to just

C++

```
m.def("module_level_function",
    [](...) {return module_level_function(...);},
    py::doc(R"--(Write multi-line comments

using a raw string.  Even " will appear correct!
)--"));
```

Part I

Bibliography and Appendices

Bibliography

Golub, G. H., and C. F. V. Loan, *Matrix computations*, Johns Hopkins series in the mathematical sciences ; 3, 2nd ed., Hopkins Univ. Press, 1991.

Moler, C. B., and C. F. V. Loan, Nineteen dubious ways to compute the exponential of a matrix, *SIAM Review*, *20*, 801–836, 1979.

Press, W., S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical recipes in C*, 2nd ed., Cambridge University Press, 1997.

Part II

Index

Index