ARTS Developer Guide

edited by

Stefan Buehler and Patrick Eriksson

October 7, 2020
ARTS Version 2.3.1287 (git: cef185d2-dirty)

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 http://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 http://www.radiativetransfer.org//docs/.

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 3.6.
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 `somethings`. 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. 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 assert)

The idea behind assert is simple. Suppose that at a certain point in your code, you expect two variables to be equal. If this expectation is a precondition that must be satisfied in order for the subsequent code to execute correctly, you must assert it with a statement like this:

```
assert(var1 == var2);
```

In general assert takes as argument a boolean expression. If the boolean expression is true, execution continues. Otherwise the `abort` system call is invoked and the program execution is stopped. If a bug prevents the precondition from being true, then you can trace the bug at the point where the precondition breaks down instead of further down in execution or not at all. The `assert` call is implemented as a C preprocessor macro, so it can be enabled or disabled at will.

In ARTS, you don't have to do this manually, as long as your source file includes `arts.h` either directly or indirectly. Instead, assertions are turned on and off with the global NDEBUG preprocessor macro, which is set or unset automatically by the `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`).

If your program is stopped by an assertion failure, then the first thing you should do is to find out where the error happens. To do this, run the program under the GDB debugger. First invoke the debugger:

```
gdb arts
```

You have to give the full path to the ARTS executable. Then set a breakpoint at the assertion failure:

```
(gdb) break __assert_fail
```

(Note the two leading underscores!) Now run the program:

```
(gdb) run
```

Instead of just exiting, under the debugger the program will be paused when the assertion fails, and you will get back the debugger prompt. Now type:

```
(gdb) where
```

to see where the assertion failure happened. You can use the `print` command to look at the contents of variables and you can use the `up` and `down` commands to navigate the stack. For more information, see the GDB documentation or type `help` at the prompt of GDB.

For ARTS, the assertion failures mostly happen inside the Tensor / Matrix / Vector package (usually because you triggered a range check error, i.e., you tried to read or write beyond array bounds). In this case the `up` command of GDB is particularly useful. If you give this a couple of times you will finally end up in the part of your code that caused the error.

Recommendation: In Emacs there is a special GDB mode. With this you can very conveniently step through your code.

Chapter 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 vectors and matrices are implemented in ARTS and how they are used. Furthermore it describes how arrays of arbitrary type can be constructed and used.

3.1 Implementation files

The Matrix and Vector classes described below reside in the files:

- `matpackI.h`
- `make_vector.h`
- `matpackI.cc`
- `make_vector.cc`

Tensors of order 3 to 7 reside in the files:

- `matpackIII.h`
- `matpackIV.h`
- `matpackV.h`
- `matpackVI.h`
- `matpackVII.h`

The template class `Array` (also described below) is implemented in the files:

- `array.h`
- `make_array.h`

History	
030807	Sparse added by Mattias Ekström.
030109	Documentation for using jokers without Range added by Stefan Buehler.
020516	Tensors added by Stefan Buehler.
011018	Created and written by Stefan Buehler.

The Sparse class is described in the file:

- `matpackII.h`

The file `test_matpack.cc` contains test cases and usage examples. For Sparse there is a separate test file, `test_sparse.cc`.

3.2 Vectors

The class `Vector` implements the mathematical concept of a vector. (Surprise, surprise.) This means that:

- A Vector contains a list of floating point values of type `Numeric`.
- A Vector can be multiplied with another Vector (scalar product), or with a Matrix.
- Sub-ranges of a Vector can easily be accessed, and used as if they were Vectors.
- Resizing a Vector is expensive and should be avoided.

3.2.1 Constructing a Vector

You can construct an object of class Vector in any of these ways:

```
Vector a;          // Create empty Vector.
Vector b(3);       // Create Vector of length 3, if
                   // created like this it will contain
                   // arbitrary values.
Vector c(3,0.0);   // Create Vector of length 3, and
                   // fill it with 0.

Vector d=c;        // Make d a copy of c.

Vector e(1,5,1);   // 1, 2, 3, 4, 5
Vector f(1,5,.5);  // 1, 1.5, 2, 2.5, 3
Vector g(5,5,-1);  // 5, 4, 3, 2, 1
Vector h{1.0,2.0,3.0};   // Creates a vector of length 3
                         // containing the values
                         // 1.0, 2.0, and 3.0.
```

The last three examples all use the same constructor, which takes the three arguments 'start', 'extent', and 'stride'. It will create a Vector containing 'extent' elements, starting with 'start', with a step of 'stride'.

3.2.2 VectorViews

An object of class `VectorView` is, like the name says, just another view on an existing Vector. It does not have its own data. This has the important consequence that it cannot be resized, since that would mess up the original Vector that the view is referring to. You can create VectorViews from Vectors using the index operator '[]', the class `Range`, and the special `joker` object. Examples:

```
Vector x{1,2,3,4,5,6,7};
VectorView a = x;                // Now a refers to the
                                 // whole of x;
VectorView b = x[Range(joker)];  // Same effect.
VectorView c = x[Range(0,2)];    // Take 2 elements of x,
                                 // starting at the
                                 // beginning,
                                 // in this case: 1,2.
VectorView d = x[Range(0,3,2)];  // In this case: 1,3,5.
VectorView e = x[Range(3,joker)]; // In this case: 4,5,6,7.
```

As you can see, most useful ways to create VectorViews involve the Range class. The general constructor to this class takes three arguments, 'start', 'extent', and 'stride'. This means that you will select 'extent' elements from the Vector, starting with index 'start', with a step-width of stride. Note that indices are 0-based, so 0 refers to the first element. The last argument, 'stride', can be omitted, in that case the default of 1 is assumed. As a special case, 'extent'==`joker` means 'to the end', and calling Range with only one argument `joker` means 'all elements'.

Usually, you will not have to use VectorView explicitly, because you can use expressions like:

```
Vector a(1,5,1);               // a = 1,2,3,4,5
Vector b = a[Range(1,3)];      // b = 2,3,4
```

However, `VectorView` and the related class `ConstVectorView` are extremely useful as the argument types of functions operating on Vectors. You should define your functions like this:

```
void silly_function(VectorView a,      // Output argument
                    ConstVectorView b  // Input argument
                                       // (read only)
                   )
{
   // Do some silly stuff with a and b.
}
```

Note that there must not be any '&' after VectorView or ConstVectorView. In other words they have to be passed by value, not by reference. This is ok, since they do not contain the actual data, so that passing by value is efficient. Passing VectorViews by reference is forbidden.

You should use these kind of arguments for all input Vectors, and also for the output if you have a function that does not resize the output Vector. This has the great advantage that you can call the function with Vector sub-ranges, e.g.,

```
Vector a(1,5,1);                 // a = 1,2,3,4,5
Vector b(3);                     // Set size of b.
silly_function(b,a[Range(0,3)]); // Call fuction with
                                 // sub-range of a.
```

An exception to this rule are workspace methods, which use conventional argument types `const Vector&` for input and `Vector&` for output.

3.2.3 What you can do with a Vector (or VectorView)

All examples below (except for the first) assume that a is a Vector or VectorView.

Resize (only for Vector, not for VectorView!):

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

This makes a a 5 element vector. The new Vector is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost. Appending to a Vector is not possible.

Get the number of elements:

```
cout << a.nelem();
```

Sum up all elements:

```
cout << a.sum();
```

Element access:

```
cout << a[3];    // Print 4th element.
a[0] = 3.5;      // Assign 3.5 to first element.
```

Note that we use 0-based indexing! Furthermore note that the operator '[]' can be also used with Range, as explained above.

Copying Vectors:

```
Vector b;
b = a;
```

In this case the size of b will be adjusted to that of a automatically. Maybe you have noticed that there is a way to formulate the example above in even shorter fashion:

```
Vector b = a;
```

The result is exactly the same. Note, though, that in this case b is *constructed* from a, not copied (see section about constructing Vectors above).

Copying in connection with views:

This one is a bit tricky. Obviously, the size of views can not be adjusted, because a view is just some selection of the underlying object. The '=' operator in this case copies the *contents*, so the sizes of the left-hand and right-hand argument must match. VectorView internally uses assertions to make sure of this. So, if you get an assertion failure one reason could be that you forgot to make the target the correct size. Here is an example:

```
b[Range(5,5,-1)] = a[Range(3,5)];  // Copy 5 elements from
                                   // a to b, reversing
                                   // the order and starting
                                   // with index 3 in a.
```

Great, isn't it?

Assigning a scalar:

```
a = 1.0;                        // Assign 1 to all elements.
```

Mathematical operators:

```
Vector a(1,3,1), b(3,1); // a = 1,2,3; b = 1,1,1
a *= 2;                  // a = 2,4,6
                         // Similarly, /=, +=, -=
a += b;                  // a = 3,5,7
                         // Similarly, -=, *=, /=
a += a;                  // a = 6,10,14
                         // So a can appear on both sides.
```

All these operate element-wise. Note, that there are no return versions of these operators (i.e., expressions like `b = a+1` are not possible). This is again for efficiency reasons. It is currently an active area of research in programming techniques how to make this kind of expression efficient. None of the available solutions works, so ARTS has to live without it.

Maximum, minimum and mean:

```
cout << max(a);
cout << min(a);
cout << mean(a);
```

Scalar product:

```
cout << a*a;
```

This is an exception to the rule not to have return versions of operators. The reason is quite obvious: The return value is only a scalar.

Arbitrary single-argument math functions:

```
Vector b(a.nelem());
transform(b,sin,a);   // b = sin(a)
transform(b,cos,b);   // b = sin(b)
                      // So b can appear on both sides.
```

The transform function operates on each element of `a` with the function you specify and puts the result in `b`. Note that the order of the arguments is swapped compared to the old function `trans` that we had in the pre-Matpack era.

3.3 Matrices

The class `Matrix` implements the mathematical concept of a matrix. (Who would have guessed this?) This means that:

- A Matrix contains floating point values of type Numeric.

- The values are arranged in rows and columns and can be accessed by indices. The first index is the row, the second the column. In other words, we use *row-major* order, similar to C, Matlab, and most math textbooks. Note, however, that some languages like FORTRAN and IDL use *column-major* order.
- A Matrix can be multiplied with a Vector, or with another Matrix.
- A sub-range of a Matrix in both dimensions (submatrix) can easily be accessed, and used as if it was just a normal matrix.
- Resizing a Matrix is expensive and should be avoided.

3.3.1 Constructing a Matrix

You can construct an object of class Matrix in any of these ways:

```
Matrix a;           // Create empty Matrix.
Matrix b(3,4);      // Create Matrix with 3 rows
                    // and 4 columns. When
                    // created like this it will contain
                    // arbitrary values.
Matrix c(3,4,0.0); // Similar, but
                    // fill it with 0.

Matrix d=c;         // Make d a copy of c.
```

3.3.2 MatrixViews

A `MatrixView` is a view on an existing Matrix, in the same way as a `VectorView` is a view on an existing Vector. Like a VectorView, a MatrixView cannot be resized and does not contain the actual data. A view is generated by using Ranges:

```
Matrix x(10,20);                  // Create 10x20 matrix.
MatrixView a = x;                 // Now a refers to the
                                  // whole of x;
MatrixView b = x(Range(joker),Range(joker));
                                  // Same effect.
MatrixView c = x[Range(0,2),Range(0,2)];
                                  // 2x2 sub-matrix.
```

You probably get the idea. Note that the second argument of Range gives the number of elements to take, not the index of the last element. See the section about Vectors for more examples how to use Range. You can use `joker`, and also the third argument of Range to select only every nth row, or column, or reverse the order of the rows or columns.

In analogy to the Vector case, you should use the two classes `MatrixView` and `ConstMatrixView` as function arguments. Please refer to the discussion in the Vector section for details. As in the case of VectorViews, all arguments of these types should be passed by value, not by reference. Also, similar to the Vector case, workspace methods are the exception, because they have to use the conventional `const Matrix&` or `Matrix&` as input/output arguments.

3.3.3 What you can do with a Matrix (or MatrixView)

All examples below (except for the first) assume that `a` is a Matrix or MatrixView.

Resize (only for Matrix, not for MatrixView!):

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

This makes `a` a 5x10 Matrix (5 rows, 10 columns). The new Matrix is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost.

Get the number of rows or columns:

```
cout << a.nrows();
cout << a.ncols();
```

Refer to a row or column:

```
Vector x = a(0,Range(joker));            // First row.
Vector y = a(Range(joker),a.ncols()-1); // Last column.
```

Of course, you can use more complicated Range expressions to refer to only parts of a row or column. However, the case that you want all elements of a given dimension is so much more common than the more sophisticated uses of the `Range` class, that it is worth to introduce a simplified notation for this case. Therefore, you are allowed to omit the `Range` and just write:

```
Vector x = a(0,joker);           // First row.
Vector y = a(joker,a.ncols()-1); // Last column.
```

Technically, expressions of this kind return the type `VectorView`. This means, they can be used in all cases where an object of that type is expected, for example with the function defined in Section 3.2.2:

```
silly_function(a(0,Range(joker)),
               a(1,Range(joker))); // Call silly_function
                                   // with first and
                                   // second row of a.
```

Element access:

```
cout << a(3,4); // Print that element.
a(0,0) = 3.5;   // Assign 3.5 to the top-left element
```

Note that we use 0-based indexing! Furthermore note that the operator '()' can be also used with one or two `Range` arguments, as explained above. To summarize:

- (`Index`,`Index`) returns `Numeric` (element access).
- (`Index`,`Range`) or (`Range`,`Index`) returns `VectorView` (row or column access).
- (`Range`,`Range`) returns `MatrixView` (sub-matrix access).

You may find it unlogical, that Matrix uses '()' for indexing, whereas Vector uses '[]'. However, using '[]' for Matrix is not possible, since it can have only one argument. On the other hand, using '()' for Vector element access seemed not a good idea, since that would break with the established use of '[]' for element access in C and C++.

Copying Matrices:

```
Matrix b;
b = a;
```

As in the case of Vectors, the '=' operator adjusts the size of the target automatically.

Copying in connection with views:

As in the case of Vectors, the '=' operator copies only the *contents* for views, so the dimensions must match. An attempt to justify this behavior has been made above in the Section about Vector. As for Vector, you can use '=' with complicated expressions. Here is a more elaborate example:

```
b(Range(0,3),Range(0,4)) =
   a(Range(10,3),Range(3,4,-1)); // Copy a row 10-12,
                                 // column 0-3
                                 // to b row 0-2,
                                 // column 0-3, reversing
                                 // the order of columns.
```

Note that in this case the dimensions must match exactly, as explained in the Section about Vector.

If you do not understand the use of Range here, refer to Section 3.2.2.

Assigning a scalar:

```
a = 1.0;                        // Assign 1 to all elements.
```

Mathematical operators:

You can use the operators '+=', '-=', '*=', and '/=', which operate element-vise, just as for Vector.

Maximum, minimum and mean:

```
cout << max(a);
cout << min(a);
cout << mean(a);
```

These operations act on the complete matrix. Hence, the output is a scalar value.

Arbitrary single-argument math functions:

The function `transform` works just like for Vector.

Transpose:

```
Matrix b = transpose(a); // Make b the transpose of a.
```

The function `transpose` creates a MatrixView, for which rows and columns are interchanged. Note, that only the way the data is accessed is changed, not the data itself. So Matrix `a` in the example above is not changed. For this reason, transposing is very efficient. You can use `transpose(a)` instead of `a` in any matrix expression practically without additional cost. (This is not strictly true, after all, the view has to be generated and passed. But that cost should be negligible except for very small matrices.)

Matrix multiplication:

```
// Matrix-Vector:
Vector b(a.nrows()), c(a.ncols());
mult(b,a,c);                           // b = a * c

// Matrix-Matrix:
Matrix d(a.nrows(),5), e(a.ncols(),5);
mult(d,a,e);                           // d = a * e
```

Note, that the result is put in the first argument, consistent with the general ARTS policy, but different from the old MTL based multiplication function. Furthermore note, that as you can see from the first example, a Vector is always considered to be a 1-column Matrix.

Important: The matrices or vectors that you give for the three arguments must not overlap, or you will get garbage. In particular, this means that

```
mult(x,y,x);            // x = y * x FORBIDDEN!!!
```

does not work. No, even worse: It works, but it gives the wrong result. The reason for this behavior is that the result is constructed in the first argument variable. If that is also an input variable it will change while it is multiplied, which will lead to a different result. There is no efficient way to detect overlap, so the only way to allow input and output arguments to be identical would be to use another internal dummy variable to store the result. However, this would be much less efficient.

Another thing: You can use transpose, of course. These two examples should obviously give the same result:

```
// Define b and c as in first example above.
mult(c,transpose(a),b);                // c = a' * b

// Vector-Matrix:
mult(transpose(c),transpose(b),a);     // c' = b' * a
```

3.4 Tensors

ARTS has tensors with rank 3 to 7. They are called `Tensor3`, `Tensor4`, `Tensor5`, `Tensor6`, `Tensor7`, and work very much like matrices, just with more dimensions. Some properties:

- A Tensor contains floating point values of type Numeric.

- The *rank* of a tensor means the number of dimensions, so a Tensor4 has 4 dimensions. Tensors of different rank are different classes. That means, the rank is fixed at compile time and cannot be changed at runtime. We will use rank and dimension as synonyms.
- The different dimensions are named:
  - Library
  - Vitrine
  - Shelf
  - Book
  - Page
  - Row
  - Column

  For example, `Tensor3 b(2,4,3)` defines a third order tensor with 2 pages, 4 rows, and 3 columns. Note that the column dimension is always last. (Incidentally, a Matrix behaves exactly like a second order tensor, except that it has some additional features.)
- A sub-range of a tensor in all dimensions (sub tensor) can easily be accessed, and used as if it was just a normal tensor.
- More importantly, you can easily access lower dimensional 'slices' of a tensor.
- Resizing a tensor is expensive and should be avoided.

3.4.1 Constructing a tensor

You can construct an object of a tensor class like this:

```
Tensor7 a;              // Create empty tensor of rank 7
Tensor3 b(2,4,3);       // 2 pages, 4 rows, 3 columns
Tensor3 c(2,4,3,0.0);   // Similar, but
                        // fill it with 0.

Tensor3 d=c;            // Make d a copy of c.
```

3.4.2 Tensor views

Tensor views work exactly like matrix and vector views. Example:

```
Tensor4 a(10,20,5,4);
Tensor3View b = a(3,Range(1,3),joker,joker);
```

If you have read the previous sections carefully, it should be clear what this expression does.

This is what is meant by slicing: You can easily create a view of a tensor that picks out an object of lower dimension. Note that you can use either an Index or a `Range` argument[1] for any of the dimensions. The dimensionality of the result will adjust accordingly, as in the example above.

Everything that was said about matrix and vector views holds also here. In particular, please always use views as function arguments.

[1]Using just `joker` is equivalent to using `Range(joker)`, as explained in Section 3.3.

3.4.3 What you can do with a tensor (or tensor view)

All examples below (except for the first) assume that `a` is a Tensor7 or Tensor7View.

Resize (only for tensors, not for views):

```
a.resize(5, 10, 4, 5, 3, 6, 8);
```

This makes `a` the requested size. The new tensor is not initialized (i.e., the contents will be unpredictable). Also, note that the previous content will be completely lost.

Get the extent of the various dimensions:

```
Index nl = a.nlibraries();
Index nv = a.nvitrines();
Index ns = a.nshelves();
Index nb = a.nbooks();
Index np = a.npages();
Index nr = a.nrows();
Index nc = a.ncols();
```

Which of these functions are available depends on the dimension of your tensor. For example, `nlibraries()` is only available for Tensor7. Note, that I took care that the first letters of the dimension names are unique, which is very convenient if you prefer short names for your variables that refer so some dimension of a tensor.

Slicing:

```
Vector x = a(0,2,1,8,3,4,joker);
// Select row 4
// on page 3
// in book 8
// on shelf 1
// in vitrine 2
// in library 0
// and copy it to the Vector x.
```

Any `Range` or Index expression is allowed in any of the arguments, of course.

Element access:

```
cout << a(3,4,0,0,0,0,0); // Print that element.
a(0,0,0,0,0,0,0) = 3.5;   // Assign 3.5 to this element.
```

Copying tensors:

Works exactly like copying matrices. Size of output argument is adjusted for Tensors, but must already have the correct size for TensorViews.

Assigning a scalar:

```
a = 1.0;                        // Assign 1 to all elements.
```

Mathematical operators:

You can use the operators ‘+=’, ‘-=’, ‘*/’, and ‘/=’, which operate element-vise, just as for Vector.

Maximum and minimum:

```
cout << max(a);
cout << min(a);
```

Arbitrary single-argument math functions:

The function `transform` works just like for Vector.

3.4.4 Making things appear larger than they are

Assume that you have written a function that performs some calculation for a Tensor5:

```
void my_function(Tensor5View x);
```

Can you call this function with a Tensor4? Yes, you can:

```
Tensor4 a;
Tensor5View b = a;                // The extent of the first
                                  // dimension of b will be 1.
my_function(b);                   // Call the function.
```

In general, you can always create a view that is one dimension bigger than what you have. The leading dimension then has extent 1. There is one important exception: If you interpret a Vector as a Matrix, the trailing dimension will be 1, not the leading dimension. This is necessary, because the vector has to act like a column vector, so that matrix-vector products work in the normal way. Of course you can use telescoping to blow up anything to Tensor7:

```
Numeric b = 3.1415;               // Just any number here.
Tensor7View bt7 =
  Tensor6View(
     Tensor5View(
        Tensor4View(
           Tensor3View(
              MatrixView(
                 VectorView(b)
              )
           )
        )
     )
  );                  // All dimensions of bt7 will be 1!
```

This kind of conversion works also implicitly. So, instead of the first example, you could have simply written:

```
my_function(a);
```

The purpose of this feature is to avoid having to make special versions of auxiliary functions for all different tensor dimensions. Use it wisely! There is very little runtime overhead for this, since the data itself is not copied.

3.4.5 Summary

This is all. In particular, we have no tensor products, since they are not needed. So, tensors are mostly used to store things, notably atmospheric field and so on. There is a set of sophisticated interpolation routines, though, which are described separately in Chapter 5.

3.5 Arrays

The template class `Array` can be used to make arrays out of anything. I do not know a good definition for 'array', but I guess anybody who has written a computer program in any programming language is familiar with the concept. Of course, it is rather similar to the concept of a Vector, just missing all the mathematical functionality like Matrix-Vector multiplication and sub-range access.

The implementation of our `Array` class is based on the STL class `std::vector`, whereas the implementation of our Vector class is done from scratch. So the two implementations are completely independent. Nevertheless, I tried to make `Array` behave consistently with Vector, as much as possible. There are a number of important differences, though, hopefully sufficiently explained in this part. A short summary of important differences:

- An Array can contain elements of any type, whereas a Vector always contains elements of type Numeric.
- No mathematical functionality for Array (no sub-ranges (nothing like VectorView); no +=, -=, *=, /=; no scalar product; no `transform` function; no `mult` function; no `transpose` function).
- On the other hand, resizing (for example adding to the end) of an Array is ok. (See the `push_back` method below.) It is still rather expensive, though, at least for large Arrays.

3.5.1 Constructing an Array

You can construct an object of an Array class like this:

```
Array<Index>  a;         // Empty Array of class Index.

Array<String> b(5);      // String Array with 5
                         // elements. Without initialization,
                         // elements contain random values.
Array<String> c(5,"x"); // The same, but fill with "x".

Array<Index>  d=a;       // Make d a copy of a;
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.

3.5.2 What you can do with an Array

All examples below assume that `a` is an ArrayOfString.

Resize:

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

This adjusts the size of `a` to 5. Resizing is more efficiently implemented than for Vector, but still expensive.

Get the number of elements:

```
cout << a.nelem();  // Just as for Vector.
```

In particular, note that the return type of this method is Index, just as for Vector. This is an extension compared to std::vector, which just has a method `size()` that returns the positive integer type `size_t`.

Element access:

```
cout << a[3];    // Print 4th element.
a[0] = "Hello"; // Assign string "Hello" to first element.
```

In other words, this works just like for Vector.

Copying Arrays:

This works also the same as for Vector. The size of the target must match! In this respect, I have modified the behavior with respect to the underlying std::vector, which has different copy semantics.

Assigning a scalar of the base type:

```
a = "Hello";     // Assign string "Hello" to all elements.
```

Append to the end:

```
a.push_back("Hello"); // Adds this new element at the
                      // end of a.
```

This can be an expensive operation, especially for large Arrays. Therefore, use it with care. Actually, the `push_back` method comes from the `std::vector` class that Array is based on. You can do a lot more with `std::vector`, all of which also works with `Array`. However, to explain the Standard Template Library is beyond the scope of this text. You can read about it in C++ or even dedicated STL textbooks.

3.6 Sparse matrices

The class `Sparse` implements the mathematical concept of a matrix, same as Matrix does, but the data is stored in a different manner. Sparse offers a memory saving storage when most of the matrix is filled with zeros. This means that:

- A Sparse contains floating point values of type Numeric.
- The values are arranged in rows and columns in the same ways as for ordinary matrices, in *row-major* order.
- A Sparse can be multiplied with a Vector, a Matrix or with another Sparse.
- There exist no views for Sparse.
- Resizing a Sparse is expensive and should be avoided.

To calculate the maximum number of non-zero elements for efficient storage, take the product of number of columns and number of rows, subtract the number of columns plus one and then divide by two, $(nnz \leq 0.5 \times (ncols \times nrows - (ncols + 1))$.

3.6.1 Constructing a Sparse

You can construct an object of class Sparse in any of these ways:

```
Sparse a;           // Create empty Sparse.
Sparse b(3,4);      // Create Sparse with 3 rows
                    // and 4 columns. When
                    // created like this it will
                    // contain only zeros, i.e.
                    // be an empty Sparse.

Sparse d=c;         // Make d a copy of c.
```

3.6.2 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.

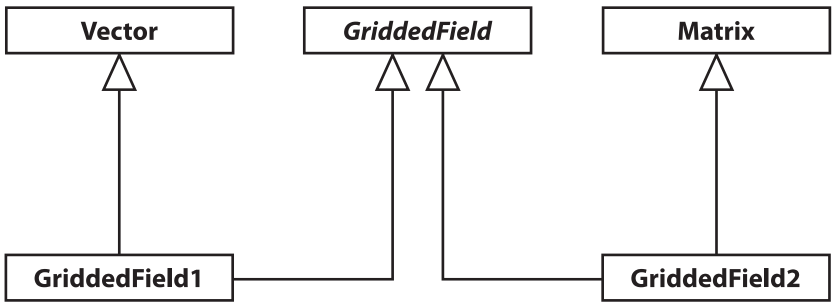

Figure 4.1: UML diagram of gridded field inheritance.

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.

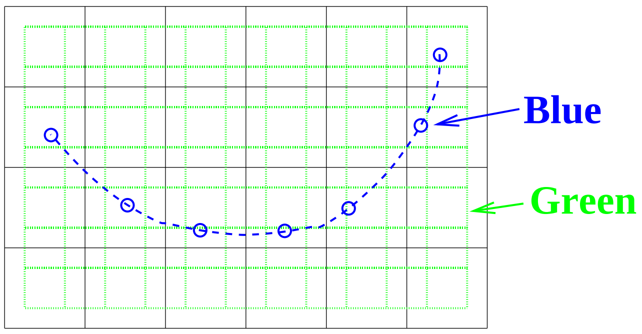

Figure 5.1: The two different types of interpolation. Green (dotted): Interpolation to a new grid, output has same dimension as input, in this case 2D. Blue (dashed): Interpolation to a sequence of points, output is always 1D.

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).

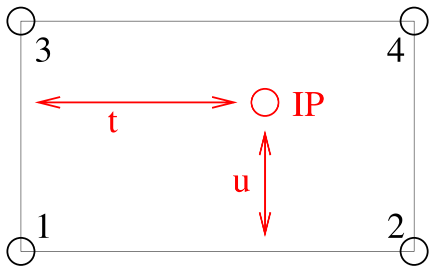

Figure 5.2: The grid square for 2D interpolation. The numbers 1...4 mark the corner points, IP is the interpolation point, t and u are the fractional distances in the two dimensions.

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 two files

- `interpolation_poly.h`
- `interpolation_poly.cc`

We define interpolation order O as the order of the polynomial that is used. Linear interpolation, the ARTS standard case, corresponds to $O = 1$. $O = 2$ is quadratic interpolation, $O = 3$ cubic interpolation. The number of interpolation points (and weights) for a 1D interpolation is $O + 1$ for each point in the new grid. So, linear interpolation uses 2 points, quadratic 3, and cubic 4.

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 `GridPosPoly`, instead of `GridPos`. A `GridPosPoly` object contains grid indices and interpolation weights for all interpolation points. For each point in the new grid, there are $O + 1$ indices and $O + 1$ weights.

The reason why we store all interpolation point indices, and not only the index of the first point, is to allow correct handling of circular interpolation, for example in scattering phase function ϕ angle. If the angle goes from 0 to 360°, then points just below 360 should be used in interpolations to points just above 0, so the indices to use are not contiguous in memory. Functions to handle this are not yet implemented, but this should be a relatively simple matter.

In contrast to `GridPos`, `GridPosPoly` stores weights `w` rather than fractional distances `fd`. For the linear case:

```
  w[0] = fd[1]
  w[1] = fd[0]
```

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

The weights are calculated according to section 3.1, eq. 3.1.1 of [*Press et al.*, 1997]. These are for the 1D case. For 2D and higher dimensional cases, the weights of the individual dimensions have to be multiplied, just as in the linear interpolation case.

Instead of `gridpos`, you have to use the function `gridpos_poly` for higher order interpolation. It works exactly like `gridpos`, but has an additional argument that gives the interpolation order O.

After setting up the `GridPosPoly` object with `gridpos_poly`, you have to call `interpweights` and `interp`, exactly as in the linear case. (The actual functions used are not the same, since the name is overloaded. The `interpweights` and `interp` functions for use with `GridPosPoly` are implemented in `interpolation_poly.cc`.) So, a complete interpolation chain involves:

```
gridpos_poly
interpweights
interp
```

For $O = 1$ the result of the interpolation chain will be the same as for the linear interpolation routines. Below is a simple complete example, taken from the file `test_interpolation.cc` in the arts source directory:

```
void test08()
{
  cout << "Very simple interpolation case for the "
       << "new higher order polynomials.\n";

  Vector og(1,5,+1);            // 1, 2, 3, 4, 5
  Vector ng(2,5,0.25);          // 2.0, 2,25, 2.5, 2.75, 3.0
  cout << "Original grid:\n" << og << "\n";
  cout << "New grid:\n" << ng << "\n";

  // To store the grid positions:
  ArrayOfGridPosPoly gp(ng.nelem());

  Index order=2;                // Interpolation order.

  gridpos_poly(gp,og,ng,order);
  cout << "Grid positions:\n" << gp;

  // To store interpolation weights:
  Matrix itw(gp.nelem(),order+1);
  interpweights(itw,gp);

  cout << "Interpolation weights:\n" << itw << "\n";

  // Original field:
  Vector of(og.nelem(),0);
  of[2] = 10;                   // 0, 0, 10, 0, 0

  cout << "Original field:\n" << of << "\n";

  // Interpolated field:
  Vector nf(ng.nelem());

  interp(nf, itw, of, gp);

  cout << "New field (order=" << order << "):\n" << nf << "\n";

  cout << "All orders systematically:\n";
  for (order=1; order<5; ++order)
    {
      gridpos_poly(gp,og,ng,order);
      itw.resize(gp.nelem(),order+1);
      interpweights(itw,gp);
      interp(nf, itw, of, gp);

      cout << "order " << order << ": ";
      for (Index i=0; i<nf.nelem(); ++i)
        cout << setw(8) << nf[i] << " ";
      cout << "\n";
    }
}
```

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 `gridpos_poly` (one for each interpolation dimension)
2. `interpweights`

3. `interp`

If you are interested in how the functions really work, look in file `interpolation.cc` or `interpolation_poly.cc`. The documentation there is quite detailed. When you are using interpolation, you should always give some thought to whether you can re-use grid positions or even interpolation weights. This can really save you a lot of computation time. For example, if you want to interpolate several fields — which are all on the same grids — to some position, you only have to compute the weights once.

Chapter 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 :

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

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

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

In spherical coordinates we can write:

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

A double integration can be splitted into two single integrations:

$$\begin{aligned} S &= \int_0^{\pi}\left(\int_0^{2\pi} f(\theta,\phi)\sin\theta\mathrm{d}\phi\right)\mathrm{d}\theta && (6.5)\\ &= \int_0^{\pi} g(\theta)\mathrm{d}\theta && (6.6)\end{aligned}$$

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.

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

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

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

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 **A** with dimension n into a product of a lower triangular matrix **L** (has elements only on the diagonal elements and below) and an upper triangular matrix **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 $(\mathbf{K})^{-\mathbf{1}} \cdot \mathbf{b}$, 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 **A** and the parameter q. As output it gives the matrix **F** which is defined above. The following example shows how to use the `matrix_exp` function:

- Initialize **A** and assign values:
  ```
  Matrix A(3,3);
  A(1,1) = 45;
  A(1,2) = 3;
  ...
  ```
- Initialize **F**:
  ```
  Matrix F(3,3);
  ```
- Give a paramater for the accuracy:
  ```
  Index q=6;
  ```
- Call the matrix exponential function:
  ```
  matrix_exp(F,A,q);
  ```
- Print the result:
  ```
  cout << "exp(A) = " << F;
  ```

Part 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