GammaLib

User Manual

Version 0.5.0
11 February 2012

Jürgen Knödlseder
Institut de Recherche en Astrophysique et Planétologie (IRAP)
knodlseder@cesr.fr
http://www.irap.omp.eu/

Note to the user

This software has been written to analyse gamma-ray data. Particular care has been taken in making the software user friendly and well documented. If you appreciated this effort, and if this software and User Manual were useful for your scientific work, the author would appreciate a corresponding acknowledgment in your published work.

Contents

1 Introduction **1**

1.1 Scope 1

1.2 Overview 1

2 Getting `GammaLib` **3**

2.1 Before you start 3

2.1.1 Installing `cfitsio` 3

2.1.2 Installing `readline` 4

2.2 Installing `GammaLib` 5

2.2.1 Downloading `GammaLib` 5

2.2.2 Configuring `GammaLib` 5

2.2.3 Building `GammaLib` 8

2.2.4 Testing `GammaLib` 8

2.2.5 Installing `GammaLib` 8

2.2.6 Setting up the `GammaLib` environment 9

2.2.7 Generating the reference documentation 9

2.3 Getting support 10

2.4 Known problems 10

3 Getting started with `GammaLib` **11**

3.1 A quick `GammaLib` tutorial 11

3.2 Using `GammaLib` from Python 11

3.3 Programming guidelines 11

3.4 Frequently asked questions 11

4 `GammaLib` modules **12**

4.1 What is in this section? 12

4.2 Observation handling (`obs`) 13

4.2.1 Overview 13

4.2.2 Describing observations using XML 14

4.2.3 Setting up and using a calibration database 14

4.2.4 Times in `GammaLib` 14

4.2.5 Energies in `GammaLib` 14

4.2.6 Regions of Interest 14

4.3 Model handling (`model`) 14

4.4 Sky maps and sky coordinates (`sky`) 14

4.5 Creation of ftools applications (`app`) 14
4.6 Numerical methods (`numerics`) 14
4.7 Linear algebra (`linalg`) 14
4.8 Optimizers (`opt`) 14
4.9 Support functions and classes (`support`) 14
4.10 FITS file interface (`fits`) 14
4.11 XML file interface (`xml`) 14

5 Instrument-specific interfaces 15
5.1 CTA interface (`cta`) 15
5.2 *Fermi*-LAT interface (`cta`) 15
5.3 Multi-wavelength interface (`mwl`) 15

6 Under the hood 16
6.1 Registry classes 16

7 Users manual 17
7.1 Linear algebra 17
7.1.1 Vectors 17
7.1.2 Matrixes 19

8 Glossary 27

9 Index 28

1 Introduction

1.1 Scope

This User Manual provides a complete description of the `GammaLib` C++ library. It is equally well suited for beginners as well as experienced users.

1.2 Overview

`GammaLib` is a self-contained, instrument independent, open source, multi-platform C++ library that implements all code required for the scientific analysis of astronomical gamma-ray data. `GammaLib` works with high-level data, which are data that are already calibrated and that generally come in form of event lists or histograms.

`GammaLib` does not rely on any third-party software, with the only exception of the `cfitsio` library from HEASARC that is used to implement the FITS interface. This makes `GammaLib` basically independent of any other software package, increasing the maintainability and enhancing the portability of the library.

`GammaLib` potentially supports any gamma-ray astronomy instrument. Large parts of the code treat gamma-ray observations in an abstract representation, and do neither depend on the characteristics of the employed instrument, nor on the particular formats in which data and instrument response functions are delivered. Instrument specific aspects are implemented as isolated and well defined modules that interact with the rest of the library through a common interface. This philosophy also enables the joint analysis of data from different instruments, providing a framework that allows for consistent broad-band spectral fitting or imaging.

`GammaLib` source code is freely available under the GNU General Public license version 3. The latest source code can be downloaded from `https://sourceforge.net/projects/gammalib/` which also provides bug trackers and mailing lists. Further information and documentation on `GammaLib` can be found on `http://gammalib.sourceforge.net/`. The present document applies to `GammaLib` version 0.5.0.

`GammaLib` is designed to compile on any POSIX compliant platform. So far, `GammaLib` has been successfully compiled and tested on Mac OS X, OpenBSD, OpenSolaris (using the gcc compiler) and many Linux flavours. Pre-packed binary versions of the code are also available for Mac OS X. For known problems with specific platforms, please refer to section 2.4.

`GammaLib` makes heavily use of C++ classes. Instrument independency is achieved by using abstract virtual base classes, which are implemented as derived classes in the instrument specific modules.

`GammaLib` is organized into four software layers, each of which comprises a number of modules (see Fig. 1; the quoted module names correspond to the folders in the source code distribution):

- **High-level analysis support** implements classes needed for the instrument independent high-level analysis of gamma-ray data, enabling the joint multi-instrument spectral and spatial fitting by forwards folding of parametric source and background models. This layer comprises modules for instrument independent handling of observations (`obs`), sky and background models (`model`), sky maps and sky coordinates (`sky`), and for the generation of analysis executables in form of ftools (`app`).
- **Instrument specific modules** support the analysis of data from the Cherenkov Telescope Array (`cta`), the *Fermi*/LAT telescope (`lat`), and any multi-wavelength information in form of spectral data points (`mwl`).
- **Core services** comprise modules for numerical computations (`numerics`), linear algebra (`linalg`), parameter optimization (`opt`), and support functions and classes (`support`).
- **Interfaces** are implemented for reading and writing of FITS files (`fits`) and XML files (`xml`).

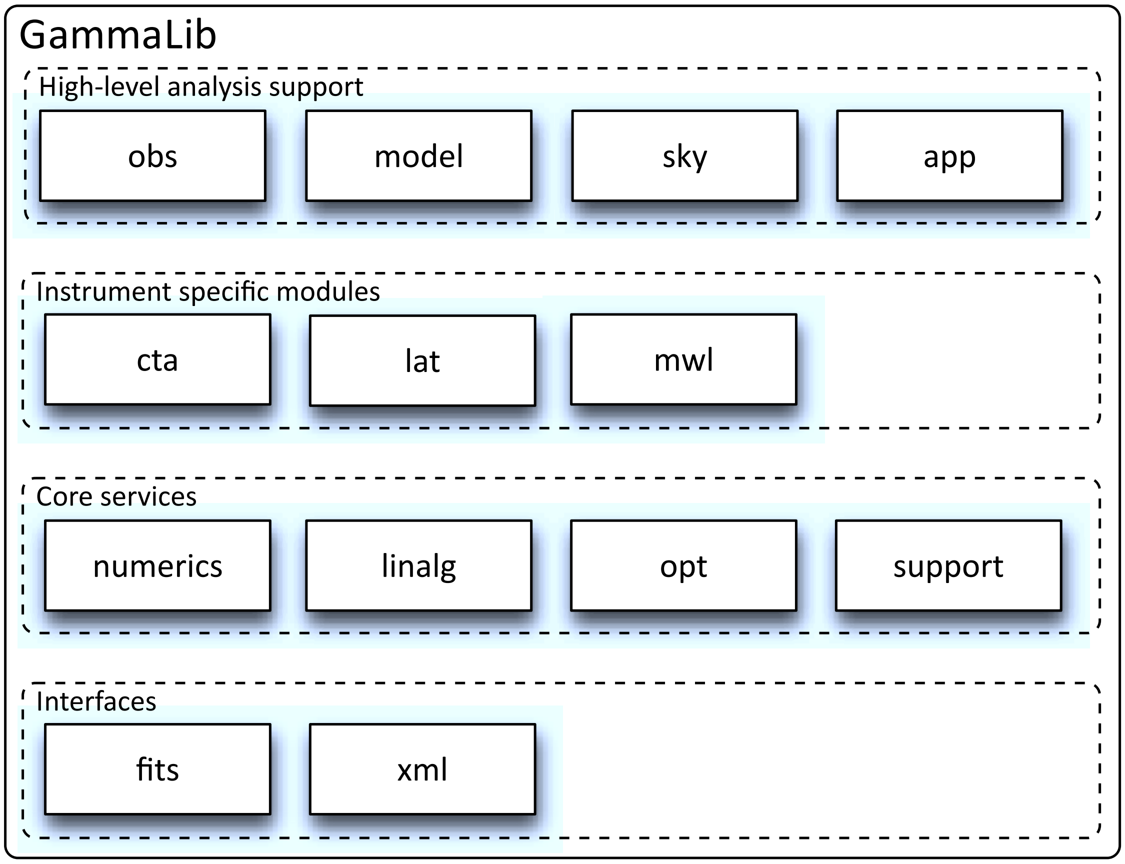

Figure 1: Organisation of `GammaLib` into software layers and modules.

`GammaLib` can be used as C++ application program interface (API) or as a Python module (provided that Python is installed on your system). The `GammaLib` Python bindings were built using `swig` version 2.0.4 (`http://www.swig.org/`), and are shipped together with the source code. This enables using all `GammaLib` functionalities from within Python.

The development of `GammaLib` has been initiated by scientists from IRAP (Institut de Recherche en Astrophysique et Planétologie), an astrophysics laboratory of CNRS and of the University Paul Sabatier situated in Toulouse, France. `GammaLib` is based on past experience gained in developing software for gamma-ray space missions, such as the COMPTEL telescope aboard *CGRO*, the SPI telescope aboard *INTEGRAL*, and the LAT telescope aboard *Fermi*. Initial elements of `GammaLib` can be found in the `spi_toolslib` that is part of the Off-line Science Analysis (OSA) software distributed by ISDC for the science analysis of *INTEGRAL* data. The development of `GammaLib` is nowadays mainly driven by the advances in ground-based gamma-ray astronomy, and in particular by the development of the CTA observatory.

2 Getting `GammaLib`

2.1 Before you start

The procedure for building and installing `GammaLib` is modeled on GNU software distributions. You do not need to have system administrator privileges to compile and to install `GammaLib`.

You will need the following to build the software:

- About 100 MB of free disk space.
- An ANSI C++ compiler. We recommend building `GammaLib` with the GNU g++ compiler.
- GNU make
- The `cfitsio` library for FITS file support together with the developer package that includes the `cfitsio.h` header.

Note that `GammaLib` compiles also in the absence of the `cfitsio` library, yet without `cfitsio`, FITS file reading or writing is not supported.

Furthermore, the following optional packages are supported but are not required to compile `GammaLib`:

- Python, including the Python developer package that includes the `Python.h` header file. If Python is present, the `GammaLib` Python module will be built and installed, allowing to script all `GammaLib` functionalities from within Python.
- `readline`, including the `readline` developer package that provides the `readline.h` header file. If `readline` is present, the packages are used to enhance the user interface when entering parameters for ftools applications (see section 4.5).

If you plan to modify or to extend the `GammaLib` source code, the following software is also required on your system:

- GNU `autoconf` (`http://www.gnu.org/software/autoconf/`) and `automake` (`http://www.gnu.org/software/automake/`) is needed to rebuild the `configure` script and `Makefile.am` resource files following configuration modifications.
- `swig` (`http://www.swig.org/`) is needed to rebuild the Python wrappers following Python interface modifications. Make sure to install the latest `swig` version (2.0.4) to guarantee the largest possible compatibility of the Python wrappers.
- Doxygen (`http://www.doxygen.org/`) is needed to rebuild the software reference manual following code modifications.

The following sections provide some information about the installation of `cfitsio` and `readline`.

2.1.1 Installing `cfitsio`

HEASARC's `cfitsio` library comes on many Linux distributions as pre-compiled binary, and there are good chances that the package is already installed on your system. For Mac OS X, `cfitsio` can be installed from Mac Ports. If you use a pre-compiled binary, make sure that also the developer package is installed on your system. The developer package provides the `cfitsio.h` header file which is needed to compile in FITS file support in `GammaLib`. Please refer to the documentation of your Linux distribution to learn how to install pre-compiled binary packages (note that the installation of pre-compiled binary packages usually requires system administrator privileges).

If you need (or prefer) to install `cfitsio` from source, you can download the latest source code from `http://heasarc.gsfc.nasa.gov/fitsio`. Detailed installation instructions can also be found on this site. We recommend that you install `cfitsio` as a shared library in the same directory in which you will install `GammaLib`, so that `cfitsio` is automatically found by the `GammaLib` `configure` script. By default, `GammaLib` gets installed into the directory `/usr/local/gamma`.

You can install version 3.290 of `cfitsio` (the latest version that was available during writing this manual) by executing the following command sequence (`>` denotes the UNIX shell prompt):

```
> wget ftp://heasarc.gsfc.nasa.gov/software/fitsio/c/cfitsio3290.tar.gz
> tar xfz cfitsio3290.tar.gz
> cd cfitsio
> ./configure --prefix=/usr/local/gamma
> make shared
> sudo make install
```

The `--prefix=/usr/local/gamma` option specifies the directory into which `cfitsio` gets installed. We choose here the default `GammaLib` installation directory `/usr/local/gamma`. As this directory is a system directory, we need to use `sudo` for installation. If you decide to install `cfitsio` into a local directory which is owned by yourself, it is sufficient to type `make install` to install the library.

2.1.2 Installing `readline`

`readline` comes on all Linux distributions that are known to us as pre-compiled binary, and it is almost certain that `readline` is already installed on your system. Very often, however, the `readline` developer package that provides the `readline.h` header file is not installed, and you need to install this package yourself to enable `readline` support for `GammaLib`. Please refer to the documentation of your Linux distribution to learn how to install pre-compiled binary packages (note that the installation of pre-compiled binary packages usually requires system administrator privileges).

If you need (or prefer) to install `readline` from source, you need also to install the `ncurses` library that is required by `readline`. Here is the command line sequence that will install `ncurses` (version 5.9) and `readline` (version 6.2) in the `GammaLib` default install directory `/usr/local/gamma` from source (`>` denotes the UNIX shell prompt):

```
> wget http://ftp.gnu.org/gnu/ncurses/ncurses-5.9.tar.gz
> tar xfz ncurses-5.9.tar.gz
> cd ncurses-5.9
> ./configure --prefix=/usr/local/gamma
> make
> sudo make install
> cd ..
> wget http://ftp.gnu.org/gnu/readline/readline-6.2.tar.gz
> tar xfz readline-6.2.tar.gz
> cd readline-6.2
> ./configure --prefix=/usr/local/gamma
> make
> sudo make install
```

Note that `sudo` is only needed if you are not the owner of the install directory.

2.2 Installing GammaLib

2.2.1 Downloading GammaLib

To get the latest version of GammaLib, please visit the site https://sourceforge.net/projects/gammalib/. The code can be downloaded from this site by clicking on the download button. Alternatively, the code can be downloaded and unpacked from the UNIX prompt using (> denotes the UNIX shell prompt):

```
> wget --no-check-certificate https://downloads.sourceforge.net/project/gammalib/
gammalib/gammalib-00-05-00.tar.gz
> tar xfz gammalib-00-05-00.tar.gz
```

The GammaLib source code can also be cloned using git. This method is recommended if you plan to contribute to the development of the GammaLib library. Assuming that git is installed on your system, you may clone GammaLib using:

```
> git clone git://gammalib.git.sourceforge.net/gitroot/gammalib/gammalib
```

2.2.2 Configuring GammaLib

Once you've downloaded and uncompressed GammaLib, step into the GammaLib source code directory and type

```
> ./configure
```

to configure the library for compilation. Make sure that you type ./configure and not simply configure to ensure that the configuration script in the current directory is invoked and not some other system-wide configuration script.

If you would like to install GammaLib in a different directory, use the optional --prefix argument during the configuration step. For example

```
> ./configure --prefix=/home/myname/gamma
```

installs GammaLib in the gamma directory that will be located in the user's myname home directory. You can obtain a full list of configuration options using

```
> ./configure --help
```

If configuration was successful, the script will terminate with printing information about the configuration. This information is important in case that you encounter installation problems, and may help you to diagnose the problems. The typical output that you may see is as follows:

```
  GammaLib configuration summary
  ==============================
  * FITS I/O support            (yes)   /usr/local/gamma/lib /usr/local/gamma/include
  * Readline support            (yes)
  * Ncurses support             (yes)
  * Python                      (yes)
  * Python.h                    (yes)
  * swig                        (yes)
  * Make Python bindings        (yes)
  * Multiwavelength interface    (yes)
  * Fermi-LAT interface          (yes)
  * CTA interface                (yes)
  * Doxygen                      (yes)   /usr/local/bin/doxygen
  * Perform NaN/Inf checks       (yes)   (default)
  * Perform range checking       (yes)   (default)
  * Optimize memory usage        (yes)   (default)
  - Compile in debug code        (no)    (default)
  - Enable code for profiling    (no)    (default)
```

The script informs whether `cfitsio` has been found (and eventually also gives the directories in which the `cfitsio` library and the header file resides), whether `readline` and `ncurses` have been found, and whether Python including the `Python.h` header file is available. Although none of these items is mandatory, we highly recommend to install `cfitsio` to support FITS file reading and writing (see section 2.1.1), and to install Python to enable `GammaLib` scripting.

If `cfitsio` is installed on your system but not found by the `configure` script, it may be located in a directory that is not known to the `configure` script. By default, `configure` will search for `cfitsio` (in the given order) in the `GammaLib` install directory, in all standard paths (e.g. `/usr/lib`, `/usr/local/lib`, ...), and in some system specific locations, including `/opt/local/lib` for Mac OS X. Assuming that you installed `cfitsio` on your system in the directory `/home/myname/cfitsio`, you may explicitly specify this location to `configure` using the `LDFLAGS` and `CPPFLAGS` environment variables:

```
> ./configure LDFLAGS=-L/home/myname/cfitsio/lib CPPFLAGS=-I/home/myname/cfitsio/include
```

Here, `LDFLAGS` specifies the path where the shared `cfitsio` library is located, while `CPPFLAGS` specifies the path where the `cfitsio.h` header file is located. Note that `-L` has to prefix the library path and that `-I` has to prefix the header file path. With the same method, you may specify any non-standard location for the `readline` and `ncurses` libraries.

The configuration script also checks for the presence of `swig`, which is used for building the Python wrapper files. Normally, `swig` is not needed to create the Python bindings as the necessary wrapper files are shipped with the `GammaLib` source code. If you plan, however, to modify or to extend the Python interface, you will need `swig` to rebuild the Python wrappers following changes to the interface.

The configuration summary informs also about all instrument dependent interfaces that will be compiled into the `GammaLib` library. By default, all available interfaces (multi-wavelength, *Fermi*-LAT, and CTA) will be compiled into `GammaLib`. If you wish to disable a particular interface, you may use the `configure` options `--without-mwl`, `--without-lat`, or `--without-cta`. For example,

```
> ./configure --without-mwl --without-lat
```

will compile `GammaLib` without the multi-wavelength and the *Fermi*-LAT interfaces. In this case, only CTA data analysis will be supported.

`GammaLib` uses Doxygen (`http://www.doxygen.org/`) for code documentation, and the latest `GammaLib` reference manual can be found at `http://gammalib.sourceforge.net/doxygen/`. In case that you want to install the reference manual also locally on your machine, Doxygen is needed to create the reference manual from the source code. Doxygen is also needed if you plan to modify or extend the `GammaLib` library to allow rebuilding the reference documentation after changes. Please read see section 2.2.7 to learn how to build and to install the reference manual locally.

Finally, there exist a number of options that define how exactly `GammaLib` will be compiled.

Several methods are able to detect invalid floating point values (either `NaN` or `Inf`), and by default, these checks will be compiled in the library to track numerical problems. If you want to disable these checks, you may specify the `--disable-nan-check` option during configuration.

Range checking is performed by default on all indices that are provided to methods or operators (such as vector or matrix element indices, sky pixels, event indices, etc.), at the expense of a small speed penalty that arises from these verifications. You may disable these range checkings by specifying the `--disable-range-check` option during configuration.

In a few places there exists a trade-off between speed and memory requirements, and a choice has to be made whether faster execution or smaller memory allocation should be preferred. By default, smaller memory allocation is preferred by `GammaLib`, but if you are not concerned about memory allocation you may specify the `--disable-small-memory` option during configuration to speed up the code.

If you develop code for `GammaLib` you may be interested in adding some special debugging code, and this debugging code can be compiled in the library by specifying the `--enable-debug` option during configuration. By default, no debugging code will be added to `GammaLib`.

Another developer option concerns profiling, which may be of interest to optimize the execution time of your code. If you would like to add profiling information to the code (which will be at the expense of execution time), you may specify the `--enable-profiling` option during configuration, which adds the `-pg` flags to the compiler. By default, profiling is disabled for `GammaLib`.

Mac OS X options. The Mac OS X environment is special in that it supports different CPU architectures (intel, ppc) and different addressing schemes (32-bit and 64-bit). To cope with different system versions and architectures, you can build a universal binary by using the option

```
> ./configure --enable-universalsdk[=PATH]
```

The optional argument `PATH` specifies which OSX SDK should be used to perform the build. By default, the SDK `/Developer/SDKs/MacOSX.10.4u.sdk` is used. If you want to build a universal binary on Mac OS X 10.5 or higher, and in particular if you build 64-bit code, you have to specify `--enable-universalsdk=/`.

A second option (which is only valid in combination with the `--enable-universalsdk`) allows to specify the kind of universal build that should be created:

```
> ./configure --enable-universalsdk[=PATH] --with-univeral-archs=VALUE
```

Possible options for `VALUE` are: `32-bit`, `3-way`, `intel`, or `all`. By default, a 32-bit build will be made.

These options are in particular needed if your Python architecture differs from the default architecture of your system. To examine the Python architecture you may type:

```
> file ‘which python‘
```

which will return the architectures that are compiled in the Python executable:

```
  i386     32-bit intel
  ppc      32-bit powerpc
  ppc64    64-bit powerpc
  x86_64   64-bit intel
```

If Python is 32-bit (`ppc`, `i386`) but the compiler produces by default 64-bit code (`ppc64`, `x86_64`), the Python module will not work. Using

```
> ./configure --enable-universalsdk=/
```

will force a universal 32-bit build which creates code for `ppc` and `i386` architectures. If on the other hand Python is 64-bit (`ppc64`, `x86_64`) but the compiler produces by default 32-bit code (`ppc`, `i386`), the option

```
> ./configure --enable-universalsdk=/ --with-univeral-archs=3-way
```

will generate a universal build which contains 32-bit and 64-bit code.

2.2.3 Building `GammaLib`

Once configured you can build `GammaLib` by typing

```
> make
```

This compiles all `GammaLib` code, including the Python wrappers, and builds the dynamic `GammaLib` library and Python module.

`GammaLib` building can profit from multi-processor or multi-core machines by performing parallel compilation of source code within the modules. You can enable this feature by typing

```
> make -j<n>
```

where `<n>` is a number that should be twice the number of cores or processors that are available on your machine.

In case that you rebuild `GammaLib` after changing the configuration, we recommend to clean the directory from any former build by typing

```
> make clean
```

prior to `make`. This will remove all existing object and library files from the source code directory, allowing for a fresh clean build of the library.

2.2.4 Testing `GammaLib`

`GammaLib` comes with an extensive unit test that allows to validate the library prior to installation. **We highly recommend to run this unit test before installing the library (see section 2.2.5).**

To run the unit test type:

```
> make check
```

This will start a test of all `GammaLib` modules by using dedicated executables which will print some progress and success information into the terminal. After completion of all tests (and assuming that all instrument dependent modules are enabled), you should see the following message in your terminal:

```
===================
All 15 tests passed
===================
```

2.2.5 Installing `GammaLib`

`GammaLib` is finally installed by typing

```
> sudo make install
```

By default, GammaLib is installed in the system directory /usr/local/gamma, hence sudo needs to be prepended to enable writing in a system-level directory. If you install GammaLib, however, in a local directory of which you are the owner, or if you install GammaLib under root, you may simply specify make install to initiate the installation process.

The installation step will copy all necessary files into the installation directory. Information will be copied in the following subdirectories:

- bin contains GammaLib environment configuration scripts (see section 2.2.6)
- include contains GammaLib header files (subdirectory gammalib)
- lib contains the GammaLib library and Python module
- share contains addition GammaLib information, such as a calibration database (subdirectory caldb), documentation (subdirectory doc), and Python interface definition files (subdirectory gammalib/swig)

2.2.6 Setting up the GammaLib environment

Before using GammaLib you have to setup some environment variables. This will be done automatically by an initialisation script that has been installed in the bin subdirectory of the install directory. Assuming that you have installed GammaLib in the default directory /usr/local/gamma you need to add the following to your $HOME/.bashrc or $HOME/.profile script on a Linux machine:

```
export GAMMALIB=/usr/local/gamma
source $GAMMALIB/bin/gammalib-init.sh
```

If you use C shell or a variant then add the following to your $HOME/.cshrc or $HOME/.tcshrc script:

```
setenv GAMMALIB /usr/local/gamma
source $GAMMALIB/bin/gammalib-init.csh
```

You then have to source your initialisation script by typing (for example)

```
> source $HOME/.bashrc
```

and all environment variables are set correctly to use GammaLib properly.

2.2.7 Generating the reference documentation

The reference documentation for GammaLib is generated directly from the source code using the Doxygen documentation system (http://www.doxygen.org/). The latest GammaLib reference manual can be found at http://gammalib.sourceforge.net/doxygen/.

The reference documentation is not shipped together with the source code as this would considerably increase the size of the tarball. In case that you want to install the reference manual also locally on your machine, you first have to create the documentation using Doxygen.

Assuming that Doxygen is available on your machine (see section 2.2.2) you can create the reference documentation by typing

```
> make doxygen
```

Once created, you can install the reference manual by typing

```
> sudo make doxygen-install
```

By default, `GammaLib` is installed in the system directory `/usr/local/gamma`, hence `sudo` needs to be prepended to enable writing in a system-level directory. If you install `GammaLib`, however, in a local directory of which you are the owner, or if you install `GammaLib` under `root`, you may simply specify `make install` to initiate the installation process.

The reference manual will be installed in form of web-browsable HTML files into the folder

```
  /usr/local/gamma/share/doc/gammalib/html/doxygen
```

You can access all web-based `GammaLib` documentation locally using `file:///usr/local/gamma/share/doc/gammalib/html/index.html` (assuming that the `GammaLib` library has been installed in the default directory `/usr/local/gamma`).

In addition, the reference manual will also be available as man pages that will be installed into

```
  /usr/local/gamma/share/doc/gammalib/man
```

To access for example the information for the `GApplication` class, you can type

```
> man GApplication
```

which then returns the synopsis and detailed documentation for the requested class.

2.3 Getting support

Any question, bug report, or suggested enhancement related to `GammaLib` should be submitted via the Tracker on `https://sourceforge.net/projects/gammalib/` or by sending an e-mail to the `gammalib-users@lists.sourceforge.net` mailing list.

2.4 Known problems

Mac OS X 10.7 (Lion) replaces the GNU g++ compiler by the clang compiler (`http://clang.llvm.org/`). `GammaLib` compiles successfully on this compiler, but it appears that exceptions thrown by `GammaLib` are not correctly catched by the application linking `GammaLib`. This seems to be a bug (or a feature) of the clang compiler.

3 Getting started with `GammaLib`

3.1 A quick `GammaLib` tutorial

3.2 Using `GammaLib` from Python

3.3 Programming guidelines

3.4 Frequently asked questions

4 `GammaLib` modules

4.1 What is in this section?

This section provides an overview over all `GammaLib` modules and their C++ classes, with particular emphasis on the relation between the classes and their basic functionalities. It describes the purpose of all C++ classes and their primary usage, as well as their underlying arithmetics. However, we do not provide a detailed description of the interface and the inner workings of each C++ class. This information is provided in the reference documentation, which can be found online at `http://gammalib.sourceforge.net/doxygen/`, or which can be installed locally on your machine (see section 2.2.7).

Each `GammaLib` module is presented in a dedicated section, following the overview shown in Fig. 1 from the top-left to the bottom right. Instrument specific modules are described in a dedicated chapter (see chapter 5). All C++ classes of a module and their relations are illustrated using a UML diagram.

To explain how to read such a diagram, we show an example for five fictive classes in Fig. 2. Our example shows a container class that contains an arbitrary number of elements which are realized by an abstract base class. Names of abstract base classes are indicated in *italic* to highlight the fact that such classes can not be instantiated. The possible number of elements that may be held by the container (in this case any number) is indicated by the cardinality `0..*` situated next to the abstract base class. Our example shows also two derived classes that inherit from the abstract base class. The second derived class is associated with a single element of some other class, indicated by the cardinality `1` next to the class. This other class is not part of the actual module, and is thus shown in grey with a dotted boundary.

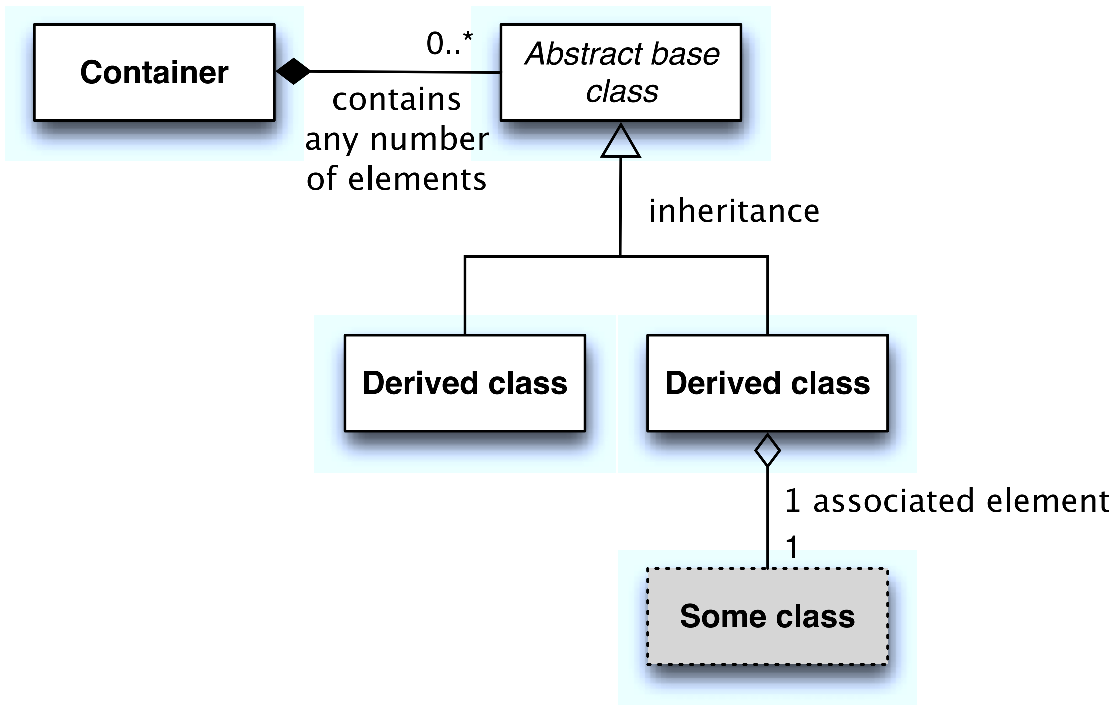

Figure 2: UML diagram illustrating the relation between five classes.

4.2 Observation handling (`obs`)

4.2.1 Overview

Figure 3 present an overview over the C++ classes of the `obs` module and their relations.

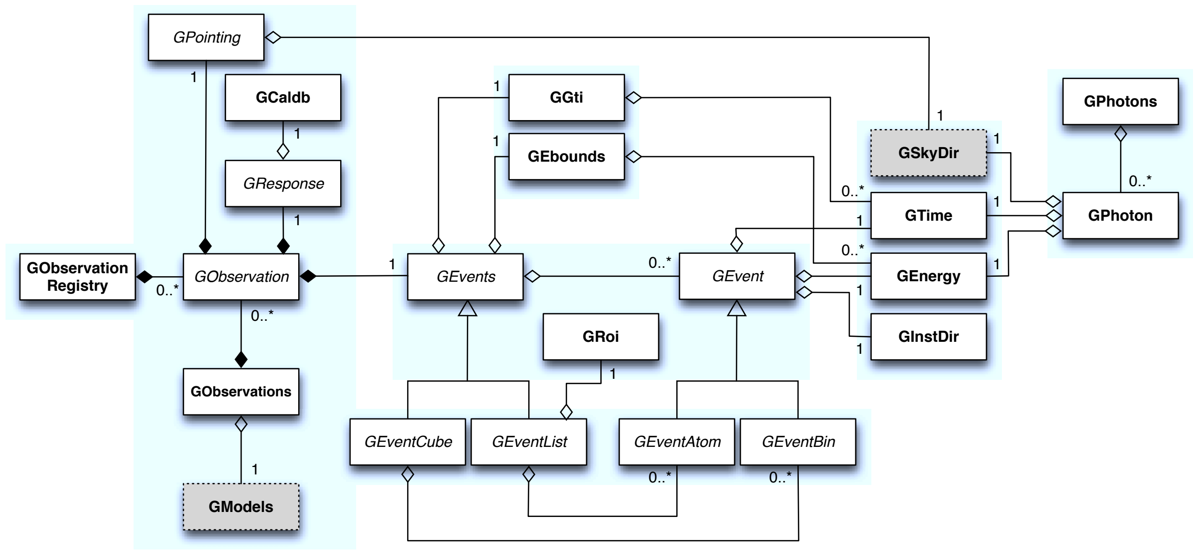

Figure 3: Relation of C++ classes in the `obs` module.

The central C++ class of the `obs` module is the abstract base class `GObservation` which defines the instrument-independent interface for a gamma-ray observation. A gamma-ray observation is defined for a single specific instrument, and describes a time period during which the instrument is in a given stable configuration that can be characterized by a single specific response function. Each gamma-ray observation is composed of events, a response function and a pointing.

Observations are collected in the C++ container class `GObservations` which is composed of a list of `GObservation` elements (the list is of arbitrary length; an empty list is a valid state of the `GObservations` class). The observation container is furthermore composed of a `GModels` model container class that holds a list of models used to describe the event distributions of the observations (see section 4.3). The `GObservations` class presents the central element of all scientific data analyses, as it combines all data and all models in a single entity.

Instrument specific implementations of `GObservation` objects are registered in the C++ registry class `GObservationRegistry` which statically collects one instance of each instrument-specific observation class that is available in `GammaLib` (see section 6.1 for a general description of registry classes).

The instrument response for a given observation is defined by the abstract base class `GResponse`. This class is composed of the C++ class `GCaldb` which implements the calibration data base that is required to compute the response function for a given instrument and observation. `GCaldb` supports the HEASARC CALDB format (`http://heasarc.nasa.gov/docs/heasarc/caldb/`), but is sufficiently general to support also other formats (see section 4.2.3 to learn how to setup and to use a calibration database).

The pointing for a given observation is defined by the abstract base class `GPointing`. This class is composed of the C++ class `GSkyDir` which implements a sky direction, which is a position on the celestial sphere (`GSkyDir` returns the position in equatorial and galactic coordinates). Note that the pointing needs not to be fixed during the observation but may evolve with time. In this case, the sky direction returned by `GPointing` will explicitly depend on time.

The events for a given observation are defined by the abstract base class `GEvents`. This class is composed of the C++ classes `GGti` and `GEbounds`. `GGti` implements so called *Good Time Intervals*, which defines the time period(s) during which the data were taken (see section 4.2.4). `GEbounds` implements so called *Energy Boundaries*, which define the energy intervals that are covered by the data (see section 4.2.5).

4.2.2 Describing observations using XML

TBW: Describe the observation XML format, and show how to handle observations using this format.

4.2.3 Setting up and using a calibration database

TBW: Describe how to setup and how to use a calibration database.

4.2.4 Times in `GammaLib`

TBW: Describe how times are implemented in `GammaLib`. This section should also handle GTIs.

4.2.5 Energies in `GammaLib`

TBW: Describe how energies are implemented in `GammaLib`. Mention that the internal energy is MeV. This section should also handle EBOUNDS.

4.2.6 Regions of Interest

TBW: Describe what a ROI is and why this is needed (unbinned analysis).

4.3 Model handling (`model`)

4.4 Sky maps and sky coordinates (`sky`)

4.5 Creation of ftools applications (`app`)

4.6 Numerical methods (`numerics`)

4.7 Linear algebra (`linalg`)

4.8 Optimizers (`opt`)

4.9 Support functions and classes (`support`)

4.10 FITS file interface (`fits`)

4.11 XML file interface (`xml`)

5 Instrument-specific interfaces

5.1 CTA interface (`cta`)

5.2 *Fermi*-LAT interface (`cta`)

5.3 Multi-wavelength interface (`mwl`)

6 Under the hood

6.1 Registry classes

TBW: Describe what a registry class is.

7 Users manual

This section provides a detailed list ...

7.1 Linear algebra

7.1.1 Vectors

General A vector is a one-dimensional array of successive `double` type values. Vectors are handled in `GammaLib` by `GVector` objects. On construction, the dimension of the vector has to be specified. In other words

```
GVector vector;                          // WRONG: constructor needs dimension
```

is not allowed. The minimum dimension of a vector is 1, i.e. there is no such thing like an empty vector:

```
GVector vector(0);                       // WRONG: empty vector not allowed
```

The correct allocation of a vector is done using

```
GVector vector(10);                      // Allocates a vector with 10 elements
```

On allocation, all elements of a vector are set to 0. Vectors may also be allocated by copying from another vector

```
GVector vector(10);                      // Allocates a vector with 10 elements
GVector another = vector;                // Allocates another vector with 10 elements
```

or by using

```
GVector vector = GVector(10);            // Allocates a vector with 10 elements
```

Vector elements are accessed using the `( )` operator:

```
GVector vector(10);                      // Allocates a vector with 10 elements
for (int i = 0; i < 10; ++i)
  vector(i) = (i+1)*10.0;                // Set elements 10, 20, ..., 100
for (int i = 0; i < 10; ++i)
  cout << vector(i) << endl;             // Dump all elements, one by row
```

The content of a vector may also be dumped using

```
cout << vector << endl;                  // Dump entire vector
```

which in the above example will put the sequence

```
10 20 30 40 50 60 70 80 90 100
```

on the screen.

Vector arithmetics Vectors can be very much handled like `double` type variables with the difference that operations are performed on each element of the vector. The complete list of fundamental vector operators is:

```
c = a + b;                        // Vector + Vector addition
c = a + s;                        // Vector + Scalar addition
c = s + b;                        // Scalar + Vector addition
c = a - b;                        // Vector - Vector subtraction
c = a - s;                        // Vector - Scalar subtraction
c = s - b;                        // Scalar - Vector subtraction
s = a * b;                        // Vector * Vector multiplication (dot product)
c = a * s;                        // Vector * Scalar multiplication
c = s * b;                        // Scalar * Vector multiplication
c = a / s;                        // Vector * Scalar division
```

where `a`, `b` and `c` are of type `GVector` and `s` is of type `double`. Note in particular the combination of `GVector` and `double` type objects in addition, subtraction, multiplication and division. In these cases the specified operation is applied to each of the vector elements. It is also obvious that only vector of identicial dimension can occur in vector operations. Dimension errors can be catched by the `try - catch` functionality:

```
try {
  GVector a(10);
  GVector b(11);
  GVector c = a + b;              // WRONG: Vectors have incompatible dimensions
}
catch (GVector::vector_mismatch &e) {
  cout << e.what() << endl;       // Dimension exception is catched here
  throw;
}
```

Further vector operations are

```
c = a;                            // Vector assignment
c = s;                            // Scalar assignment
s = c(index);                     // Vector element access
c += a;                           // c = c + a;
c -= a;                           // c = c - a;
c += s;                           // c = c + s;
c -= s;                           // c = c - s;
c *= s;                           // c = c * s;
c /= s;                           // c = c / s;
c = -a;                           // Vector negation
```

Finally, the comparison operators

```
int equal   = (a == b);           // True if all elements equal
int unequal = (a != b);           // True if at least one elements unequal
```

allow to compare all elements of a vector. If all elements are identical, the `==` operator returns true, otherwise false. If at least one element differs, the `!=` operator returns true, is all elements are identical it returns false.

In addition to the operators, the following mathematical functions can be applied to vectors:

```
    acos        atan        exp         sin         tanh
    acosh       atanh       fabs        sinh
    asin        cos         log         sqrt
    asinh       cosh        log10       tan
```

Again, these functions should be understood to be applied element wise. They all take a vector as argument and produce a vector as result. For example

```
 c = sin(a);
```

attributes the sine of each element of vector `a` to vector `c`. Additional implemented functions are

```
 c = cross(a, b);                       // Vector cross product (for 3d only)
 s = norm(a);                           // Vector norm |a|
 s = min(a);                            // Minimum element of vector
 s = max(a);                            // Maximum element of vector
 s = sum(a);                            // Sum of vector elements
```

Finally, a small number of vector methods have been implemented:

```
 int n = a.size();                      // Returns dimension of vector
 int n = a.non_zeros();                 // Returns number of non-zero elements in vector
```

7.1.2 Matrixes

General A matrix is a two-dimensional array of `double` type values, arranged in rows and columns. Matrixes are handled in `GammaLib` by `GMatrix` objects and the derived classes `GSymMatrix` and `GSparseMatrix` (see section 7.1.2). On construction, the dimension of the matrix has to be specified:

```
 GMatrix matrix(10,20);                 // Allocates 10 rows and 20 columns
```

Similar to vectors, there is no such thing as a matrix without dimensions in `GammaLib`.

Matrix storage classes In the most general case, the `rows` and `columns` of a matrix are stored in a continuous array of `rows`×`columns` memory locations. This storage type is referred to as a *full matrix*, and is implemented by the class `GMatrix`. Operations on full matrixes are in general relatively fast, but memory requirements may be important to hold all the elements. In general matrixes are stored by `GammaLib` column-wise (or in column-major format). For example, the matrix

```
    1  2  3  4  5
    6  7  8  9 10
   11 12 13 14 15
```

is stored in memory as

```
    |  1  6 11 |  2  7 12 |  3  8 13 |  4  9 14 |  5 10 15 |
```

Many physical or mathematical problems treat with a subclass of matrixes that is symmetric, i.e. for which the element `(row,col)` is identical to the element `(col,row)`. In this case, the duplicated elements need not to be stored. The derived class `GSymMatrix` implements such a storage type. `GSymMatrix` stores the lower-left triangle of the matrix in column-major format. For illustration, the matrix

```
1  2  3  4
2  5  6  7
3  6  8  9
4  7  9 10
```

is stored in memory as

```
|  1  2  3  4 |  5  6  7 |  8  9 | 10 |
```

This divides the storage requirements to hold the matrix elements by almost a factor of two.

Finally, quite often one has to deal with matrixes that contain a large number of zeros. Such matrixes are called *sparse matrixes*. If only the non-zero elements of a sparse matrix are stored the memory requirements are considerably reduced. This goes however at the expense of matrix element access, which has become now more complex. In particular, filling efficiently a sparse matrix is a non-trivial problem (see section 7.1.2). Sparse matrix storage is implemented in `GammaLib` by the derived class `GSparseMatrix`. A `GSparseMatrix` object contains three one-dimensional arrays to store the matrix elements: a `double` type array that contains in continuous column-major order all non-zero elements, an `int` type array that contains for each non-zero element the row number of its location, and an `int` type array that contains the storage location of the first non-zero element for each matrix column. To illustrate this storage format, the matrix

```
1  0  0  7
2  5  0  0
3  0  6  0
4  0  0  8
```

is stored in memory as

```
|  1  2  3  4 |  5 |  6 |  7  8 |  Matrix elements
|  0  1  2  3 |  1 |  2 |  0  3 |  Row indices for all elements
|  0          |  4 |  5 |  6    |  Storage location of first element of each column
```

This example is of course not very economic, since the total number of Bytes used to store the matrix is $8 \times 8 + (8 + 4) \times 4 = 112$ Bytes, while a full 4×4 matrix is stored in $(4 \times 4) \times 8 = 128$ Bytes (recall: a `double` type values takes 8 Bytes, an `int` type value takes 4 Bytes). For realistic large systems, however, the gain in memory space can be dramatical.

The usage of the `GMatrix`, `GSymMatrix` and `GSparseMatrix` classes is analoguous in that they implement basically all functions and methods in an identical way. So from the semantics the user has not to worry about the storage class. However, matrix element access speeds are not identical for all storage types, and if performance is an issue (as it certainly always will be), the user has to consider matrix access more carefully (see section 7.1.2).

Matrix allocation is performed using the constructors:

```
GMatrix       A(10,20);                 // Full 10 x 20 matrix
GSymMatrix    B(10,10);                 // Symmetric 10 x 10 matrix
GSparseMatrix C(1000,10000);            // Sparse 1000 x 10000 matrix

GMatrix       A(0,0);                   // WRONG: empty matrix not allowed
GSymMatrix    B(20,22);                 // WRONG: symmetric matrix requested
```

In the constructor, the first argument specifies the number of rows, the second the number of columns: `A(row,column)`. A symmetric matrix needs of course an equal number of rows and columns. And an empty matrix is not allowed. All matrix elements are initialised to 0 by the matrix allocation.

Matrix elements are accessed by the `A(row,col)` function, where `row` and `col` start from 0 for the first row or column and run up to the number of rows or columns minus 1:

```
for (int row = 0; row < n_rows; ++row) {
  for (int col = 0; col < n_cols; ++col)
    A(row,col) = (row+col)/2.0;        // Set value of matrix element
}
...
double sum2 = 0.0;
for (int row = 0; row < n_rows; ++row) {
  for (int col = 0; col < n_cols; ++col)
    sum2 *= A(row,col) * A(row,col);   // Get value of matrix element
}
```

The content of a matrix can be visualised using

```
cout << A << endl;                     // Dump matrix
```

Matrix arithmetics The following description of matrix arithmetics applies to all storage classes (see section 7.1.2). The following matrix operators have been implemented in `GammaLib`:

```
C = A + B;                             // Matrix Matrix addition
C = A - B;                             // Matrix Matrix subtraction
C = A * B;                             // Matrix Matrix multiplication
C = A * v;                             // Matrix Vector multiplication
C = A * s;                             // Matrix Scalar multiplication
C = s * A;                             // Scalar Matrix multiplication
C = A / s;                             // Matrix Scalar division
C = -A;                                // Negation
A += B;                                // Matrix inplace addition
A -= B;                                // Matrix inplace subtraction
A *= B;                                // Matrix inplace multiplications
A *= s;                                // Matrix inplace scalar multiplication
A /= s;                                // Matrix inplace scalar division
```

The comparison operators

```
int equal   = (A == B);                // True if all elements equal
int unequal = (A != B);                // True if at least one elements unequal
```

allow to compare all elements of a matrix. If all elements are identical, the `==` operator returns true, otherwise false. If at least one element differs, the `!=` operator returns true, is all elements are identical it returns false.

Matrix methods and functions A number of methods has been implemented to manipulate matrixes. The method

```
A.clear();                             // Set all elements to 0
```

sets all elements to 0. The methods

```
int rows = A.rows();                   // Returns number of rows in matrix
int cols = A.cols();                   // Returns number of columns in matrix
```

provide access to the matrix dimensions, the methods

```
double sum = A.sum();                  // Sum of all elements in matrix
double min = A.min();                  // Returns minimum element of matrix
double max = A.max();                  // Returns maximum element of matrix
```

inform about some matrix properties. The methods

```
GVector v_row    = A.extract_row(row); // Puts row in vector
GVector v_column = A.extract_col(col); // Puts column in vector
```

extract entire rows and columns from a matrix. Extraction of lower or upper triangle parts of a matrix into another is performed using

```
B = A.extract_lower_triangle();        // B holds lower triangle
B = A.extract_upper_triangle();        // B holds upper triangle
```

`B` is of the same storage class as `A`, except for the case that `A` is a `GSymMatrix` object. In this case, `B` will be a full matrix of type `GMatrix`.

The methods

```
A.insert_col(v_col,col);               // Puts vector in column
A.add_col(v_col,col);                  // Add vector to column
```

inserts or adds the elements of a vector into a matrix column. Note that no row insertion routines have been implemented (so far) since they would be less efficient (recall that all matrix types are stored in column-major format).

Conversion from one storage type to another is performed using

```
B = A.convert_to_full();               // Converts A -> GMatrix
B = A.convert_to_sym();                // Converts A -> GSymMatrix
B = A.convert_to_sparse();             // Converts A -> GSparseMatrix
```

Note that `convert_to_sym()` can only be applied to a matrix that is indeed symmetric.

The transpose of a matrix can be obtained by using one of

```
A.transpose();                         // Transpose method
B = transpose(A);                      // Transpose function
```

The absolute value of a matrix is provided by

```
B = fabs(A);                           // B = |A|
```

Matrix factorisations A general tool of numeric matrix calculs is factorisation.

Solve linear equation `Ax = b`. Inverse a matrix (by solving successively `Ax = e`, where `e` are the unit vectors for all dimensions).

For symmetric and positive definite matrices the most efficient factorisation is the Cholesky decomposition. The following code fragment illustrates the usage:

```
GMatrix A(n_rows, n_cols};
GVector x(n_rows};
GVector b(n_rows};
...
A.cholesky_decompose();              // Perform Cholesky factorisation
x = A.cholesky_solver(b);            // Solve Ax=b for x
```

Note that once the function `A.cholesky_decompose()` has been applied, the original matrix content has been replaced by its Cholesky decomposition. Since the Cholesky decomposition can be performed inplace (i.e. without the allocation of additional memory to hold the result), the matrix replacement is most memory economic. In case that the original matrix should be kept, one may either copy it before into another `GMatrix` object or use the function

```
GMatrix L = cholesky_decompose(A);
x = L.cholesky_solver(b);
```

A symmetric and positif definite matrix can be inverted using the Cholesky decomposition using

```
A.cholesky_invert();                 // Inverse matrix using Cholesky fact.
```

Alternatively, the function

```
GMatrix A_inv = cholesky_invert(A);
```

may be used.

The Cholesky decomposition, solver and inversion routines may also be applied to matrices that contain rows or columns that are filled by zeros. In this case the functions provide the option to (logically) compress the matrices by skipping the zero rows and columns during the calculation.

For compressed matrix Cholesky factorisation, only the non-zero rows and columns have to be symmetric and positive definite. In particular, the full matrix may even be non-symmetric.

Sparse matrixes The only exception that does not work is

```
GSparseMatrix A(10,10);
A(0,0) = A(1,1) = A(2,2) = 1.0;      // WRONG: Cannot assign multiple at once
```

In this case the value `1.0` is only assigned to the last element, i.e. `A(2,2)`, the other elements will remain `0`. This feature has to do with the way how the compiler translates the code and how `GammaLib` implements sparse matrix filling. `GSparseMatrix` provides a pointer for a new element to be filled. Since there is only one such *fill pointer*, only one element can be filled at once in a statement. **So it is strongly advised to avoid multiple matrix element assignment in a single row.** Better write the above code like

```
GSparseMatrix A;
A(0,0) = 1.0;
A(1,1) = 1.0;
A(2,2) = 1.0;
```

This way, element assignment works fine.

Inverting a sparse matrix produces in general a full matrix, so the inversion function should be used with caution. Note that a full matrix that is stored in sparse format takes roughly twice the memory than a normal `GMatrix` object. If nevertheless the inverse of a sparse matrix should be examined, it is recommended to perform the analysis column-wise:

```
GSparseMatrix A(rows,cols);            // Allocate sparse matrix
GVector       unit(rows);              // Allocate vector
...
A.cholesky_decompose();                // Factorise matrix

// Column-wise solving the matrix equation
for (int col = 0; col < cols; ++col) {
  unit(col) = 1.0;                     // Set unit vector
  GVector x = cholesky_solver(unit);   // Get column x of inverse
  ...
  unit(col) = 0.0;                     // Clear unit vector for next round
}
```

Filling sparse matrixes The filling of a sparse matrix is a tricky issue since the storage of the elements depends on their distribution in the matrix. If one would know beforehand this distribution, sparse matrix filling would be easy and fast. In general, however, the distribution is not known a priori, and matrix filling may become a quite time consuming task.

If a matrix has to be filled element by element, the access through the operator

```
m(row,col) = value;
```

may be mandatory. In principle, if a new element is inserted into a matrix a new memory cell has to be allocated for this element, and other elements may be moved. Memory allocation is quite time consuming, and to reduce the overhead, `GSparseMatrix` can be configured to allocate memory in bunches. By default, each time more matrix memory is needed, `GSparseMatrix` allocates 512 cells at once (or 6144 Bytes since each element requires a `double` and a `int` storage location). If this amount of memory is not adequat one may change this value by using

```
m.set_mem_block(size);
```

where `size` is the number of matrix elements that should be allocated at once (corresponding to a total memory of $12 \times$ `size` Bytes).

Alternatively, a matrix may be filled column-wise using the functions

```
m.insert_col(vector,col);              // Insert a vector in column
m.add_col(vector,col);                 // Add content of a vector to column
```

While `insert_col` sets the values of column `col` (deleting thus any previously existing entries), `add_col` adds the content of `vector` to all elements of column `col`. Using these functions is considerably more rapid than filling individual values.

Still, if the matrix is big (i.e. severeal thousands of rows and columns), filling individual columns may still be slow. To speed-up dynamical matrix filling, an internal fill-stack has been implemented in `GSparseMatrix`. Instead of inserting values column-by-column, the columns are stored in a stack and filled into the matrix once the stack is full. This reduces the number of dynamic memory allocations to let the matrix grow as it is built. By default, the internal stack is disabled. The stack can be enabled and used as follows:

```
m.stack_init(size, entries);           // Initialise stack
...
m.add_col(vector,col);                 // Add columns
...
m.stack_destroy();                     // Flush and destory stack
```

The method `stack_init` initialises a stack with a number of `size` elements and a maximum of `entries` columns. The larger the values `size` and `entries` are chosen, the more efficient the stack works. The total amount of memory of the stack can be estimated as $12 \times$ `size` $+ 8 \times$ `entries` Bytes. If a rough estimate of the total number of non-zero elements is available it is recommended to set `size` to this value. As a rule of thumb, `size` should be at least of the dimension of either the number of rows or the number of columns of the matrix (take the maximum of both). `entries` is best set to the number of columns of the matrix. If memory limits are an issue smaller values may be set, but if the values are too small, the speed increase may become negligible (or stack-filling may even become slower than normal filling).

Stack-filling only works with the method `add_col`. Note also that filling sub-sequently the same column leads to stack flushing. In the code

```
for (int col = 0; col < 100; ++col) {
  column      = 0.0;                    // Reset column
  column(col) = col;                    // Set column
  m.add_col(column,col);                // Add column
}
```

stack flushing occurs in each loop, and consequently, the stack-filling approach will be not very efficient (it would probably be even slover than normal filling). If successive operations are to be performed on columns, it is better to perform them before adding. The code

```
column = 0.0;                           // Reset column
for (int col = 0; col < 100; ++col)
  column(col) = col;                    // Set column
m.add_col(column,col);                  // Add column
```

would be far more efficient.

A avoidable overhead occurs for the case that the column to be added is sparse. The vector may contain many zeros, and `GSparseMatrix` has to filter them out. If the sparsity of the column is known, this overhead can be avoided by directly passing a compressed array to `add_col`:

```
int     number = 5;                     // 5 elements in array
double* values = new double[number];    // Allocate values
int*    rows   = new int[number];       // Allocate row index
...
m.stack_init(size, entries);            // Initialise stack
...
for (int i = 0; i < number; ++i) {      // Initialise array
  values[i] = ...                       // ... set values
  rows[i]   = ...                       // ... set row indices
}
...
m.add_col(values,rows,number,col);      // Add array
...
m.stack_destroy();                      // Flush and destory stack
...
delete [] values;                       // Free array
delete [] rows;
```

The method `add_col` calls the method `stack_push_column` for stack filling. `add_col` is more general than `stack_push_column` in that it decides which of stack- or direct filling is more adequate. In particular, `stack_push_column` may refuse pushing a column onto the stack if there is not enough space. In that case, `stack_push_column` returns a non-zero value that corresponds to the number of non-zero elements in the vector that should be added. However, it is recommended to not use `stack_push_column` and call instead `add_col`.

The method `stack_destroy` is used to flush and destroy the stack. After this call the stack memory is liberated. If the stack should be flushed without destroying it, the method `stack_flush` may be used:

```
m.stack_init(size, entries);           // Initialise stack
...
m.add_col(vector,col);                 // Add columns
...
m.stack_flush();                       // Simply flush stack
```

Once flushed, the stack can be filled anew.

Note that stack flushing is not automatic! This means, if one trys to use a matrix for calculs without flushing, the calculs may be wrong. **If a stack is used for filling, always flush the stack before using the matrix.**

8 Glossary

`cfitsio`
Library of C and Fortran subroutines for reading and writing data files in FITS data format (`http://heasarc.gsfc.nasa.gov/fitsio/`).

FITS
Flexible Image Transport System (`http://fits.gsfc.nasa.gov/`).

FTOOLS
Collection of utility programs to create, examine, or modify data files in the FITS data format (`http://heasarc.gsfc.nasa.gov/ftools/`).

GNU
A free Unix-like operating system (`http://www.gnu.org/`).

HEASARC
High Energy Astrophysics Science Archive Research Center (`http://heasarc.gsfc.nasa.gov/`).

Python
Dynamic programming language that is used in a wide variety of application domains (`http://www.python.org/`).

9 Index