{{lastupdated_at}} by {{lastupdated_by}}

Contributing to GammaLib

This page explains how you can contribute to the development of the GammaLib library.

Prerequisits

You will need the following software installed on your system to contribute to the GammaLib development:

- C++ compiler (e.g. GNU gcc)
- Git
- autoconf
- automake
- libtool
- cfitsio
- Python
- swig (version 2.0.4 or later)

Optionally, it is useful to have:

- readline
- doxygen

If some of the software is not yet installed on your system, it is very likely that you can install it through your system's package manager. Make sure that you install the development packages of cfitsio, Python and readline as they provide the header files that are required for compilation. Alternatively, you may install the software from source. On Mac OS X, you may use the MacPorts system.

Getting the source code

GammaLib uses Git for version control. **To learn how Git is used within the GammaLib project, please familiarize yourself with the [[Git workflow for GammaLib]].**

The central GammaLib Git repository can be found at https://cta-git.irap.omp.eu/gammalib. Access to the repository is managed by the https protocol. The repository is publicly accessible for reading. Pushing (i.e. writing) requires password authentification.

GammaLib is also accessible on GitHub at the address https://github.com/gammalib/gammalib. The GitHub repository is a mirror of the central GammaLib Git repository, and is read-only. You may fork the GammaLib repository on GitHub and develop your code using this fork (read [[Git workflow for GammaLib]] to learn how).

Several branches are guaranteed to exist in the GammaLib Git repository:

- master: source code of last public code release
- devel: central development branch (trunk)
- release: branch used to stabilize the code prior to a new release
- integration: branch used for code integration

Pushing to these branches is prohibited. Only the integration and release manager(s) will write into these branches.

In addition, temporary feature or hotfix branches may exist.

The default branch from which you should start your software development is the devel branch. **devel is GammaLib's trunk.** The command

```
$ git clone https://cta-git.irap.omp.eu/gammalib
```

will automatically clone this branch from the central GammaLib Git repository.

Software developments are done in feature branches. When the development is finished, issue a pull request so that the feature branch gets merged into devel. Merging is done by the GammaLib integration manager.

Preparing GammaLib for configuration

After cloning GammaLib (see above) you will find a directory called gammalib. To learn more about the structure of this directory read [[GammaLib directory structure]].
Before this directory can be used for development, we have to prepare it for configuration. **If you're not familiar with the autotools, please read this section carefully so that you get the big picture.**

Step into the gammalib directory and prepare for configuration by typing

```
$ cd gammalib
$ ./autogen.sh
```

This will produce the following output

```
glibtoolize: putting auxiliary files in `.'.
glibtoolize: copying file `./ltmain.sh'
glibtoolize: putting macros in AC_CONFIG_MACRO_DIR, `m4'.
glibtoolize: copying file `m4/libtool.m4'
glibtoolize: copying file `m4/ltoptions.m4'
glibtoolize: copying file `m4/ltsugar.m4'
glibtoolize: copying file `m4/ltversion.m4'
glibtoolize: copying file `m4/lt~obsolete.m4'
configure.ac:74: installing './config.guess'
configure.ac:74: installing './config.sub'
configure.ac:35: installing './install-sh'
configure.ac:35: installing './missing'
```

Instead of glibtoolize you may see libtoolize on some systems. Also, some systems may produce a more verbose output.

The autogen.sh script invokes the following commands:

- glibtoolize or libtoolize
- aclocal
- autoconf
- autoheader
- automake

glibtoolize or libtoolize installs files that are relevant for the libtool utility. Those are ltmain.sh under the main directory and a couple of files under the m4 directory (see the above output).

aclocal scans the configure.ac script and generates the files aclocal.m4 and autom4te.cache. The file aclocal.m4 contains a number of macros that will be used by the configure script later.

autoconf scans the configure.ac script and generates the file configure. The configure script will be used later for configuring of GammaLib (see below).

autoheader scans the configure.ac script and generates the file config.h.in.

automake scans the configure.ac script and generates the files Makefile.in, config.guess, config.sub, install-sh, and missing. Note that the files Makefile.in will generated in all subdirectories that contain a file Makefile.am.

Configuring GammaLib

There is a single command to configure GammaLib:

```
$ ./configure
```

Note that configure is a script that has been generated previously by the autoconf step. A typical output of the configuration step is provided in the file attachment:configure.out.

By the end of the configuration step, you will see the following sequence

```
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/support/Makefile
config.status: creating src/linalg/Makefile
config.status: creating src/numerics/Makefile
config.status: creating src/fits/Makefile
config.status: creating src/xml/Makefile
config.status: creating src/sky/Makefile
config.status: creating src/opt/Makefile
config.status: creating src/obs/Makefile
config.status: creating src/model/Makefile
config.status: creating src/app/Makefile
config.status: creating src/test/Makefile
config.status: creating src/gammalib-setup
config.status: WARNING:  'src/gammalib-setup.in' seems to ignore the --datarootdir setting
config.status: creating include/Makefile
config.status: creating test/Makefile
config.status: creating pyext/Makefile
config.status: creating pyext/setup.py
config.status: creating gammalib.pc
config.status: creating inst/Makefile
config.status: creating inst/mwl/Makefile
config.status: creating inst/cta/Makefile
config.status: creating inst/lat/Makefile
config.status: creating config.h
```

Here, each file with an suffix .in will be converted into a file without suffix. For example, Makefile.in will become Makefile, or setup.py.in will become setup.py. In this step, variables (such as path names, etc.) will be replaced by their absolute values.

This sequence will also produce a header file named config.h that is stored in the gammalib root directory. This header file will contain a number of compiler definitions that can be used later in the code to adapt to the specific environment. Recall that config.h.in has been created using autoheader by scanning the configure.ac file, hence the specification of which compiler definitions will be available is ultimately done in configure.ac.

The configure script will end with a summary about the detected configuration. Here a typical example:

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

The configure script tests for the presence of required libraries, checks if Python is available, analysis whether swig is needed to build Python wrappers, signals the presence of instrument modules, and handles compile options. To learn more about available options, type

```
$ ./configure --help
```

Compiling GammaLib

GammaLib is compiled by typing

```
$ make
```

In case you develop GammaLib on a multi-processor and/or multi-core machine, you may accelerate compilation by specifying the number of threads after the -j argument, e.g.

```
$ make -j10
```

uses 10 parallel threads at maximum for compilation.
Note that GammaLib requires gmake, which on most systems is in fact aliased to make. If this is not the case on your system, use gmake explicitely. (gmake is required due to some of the instructions in pyext/Makefile.am).

When compiling GammaLib, one should be aware of the way how *dependency tracking* has been setup. Dependency tracking is a system that determines which files need to be recompiled after modifications. For standard dependency tracking, the compiler preprocessor determines all files on which a given file depends on. If one of the dependencies changes, the file will be recompiled. The problem with the standard dependency tracking is that it always compiles a lot of files, which would make the GammaLib development cycle very slow. For this reason, **standard dependency tracking has been disabled**. This has been done by adding no-dependencies to the options for the AM_INIT_AUTOMAKE macro in configure.ac. If you prefer to enable standard dependency tracking, it is sufficient to remove no-dependencies from the options. After removal, run

```
$ autoconf
$ ./configure
```

to reconfigure GammaLib with standard dependency tracking enabled.

Without standard dependency tracking, the following logic applies:

- All source code files (.cpp) that have been modified will be recompiled when make is invoked. make uses the modification date of a file to determine if recompilation is needed. Thus, to enforce recompilation of a file it is sufficient to update its modification date using touch, e.g.

  ```
  touch src/app/GPars.cpp
  ```

  enforces recompilation of the GPars.cpp file and rebuilding of the GammaLib library.

- Header files (.hpp) that have been modified will **not** lead to recompilation. This is a major hurdle, and if you think that a modification of a header file affects largely the system, it is recommended to make a full build using

  ```
  $ make clean
  $ make
  ```

  Very often, however, only the corresponding source file is affected, and it is sufficient to touch the source file to enforce its recompilation.

- SWIG file (.i) that have been modified will lead to a rebuilding of the Python interface. In fact, dependency tracking has been enabled for the SWIG files. In case you want to rebuild the entire Python interface, execute the following command sequence

  ```
  $ cd pyext
  ```

```
    $ make clean
    $ cd ..
    $ make
```

When a Git branch is changed, the files that differ will have different modification dates, hence recompilation will take place. However, as mentioned above, header file (.hpp) changes will not be tracked, which could lead to a corrupt system. It is thus recommended to recompile the full library after a change of the Git branch using

```
$ make clean
$ make
```

Running unit tests

Introduction

The GammaLib unit test suite is run using

```
$ make check
```

In case that the unit test code has not been compiled so far (or if the code has been changed), compilation is performed before testing. Full dependency tracking is implemented for the C++ unit tests, hence all GammaLib library files that have been modified will be recompiled, the library will be rebuilt, and unit tests will be recompiled (in needed) before executing the test. However, full dependency tracking is still missing for the Python interface, hence one should make sure that the Python interface is recompiled before executing the unit test (see above).

A successful unit test will terminate with

```
===================
All 20 tests passed
===================
```

For each of the unit tests, a test report in JUnit compliant XML format will be written into the gammalib/test/reports directory. These reports can be analyzed by any standard tools (e.g. Sonar).

Running a subset of the unit tests

To execute only a subset of the unit tests, type the following

```
$ env TESTS="test_python.py test_CTA" make -e check
```

This will execute the Python and the CTA unit test. You may add any combination of tests in the string.

If you'd like to repeat the subset of unit test it is sufficient to type

```
$ make -e check
```

as TESTS is still set. To unset TESTS, type

```
$ unset TESTS
```

Debugging unit tests

As GammaLib developments should be test driven, the GammaLib unit tests are an essential tool for debugging the code. If some unit test fails for an unknown reason, the gdb debugger can be used for analysis as follows. Each test executable (e.g. test_CTA) is in fact a script that properly sets the environment and then executes the test binary. Opening the test script in an editor will reveal the place where the test executable is called:

```
$ nano test_CTA
...
# Core function for launching the target application
func_exec_program_core ()
{

    if test -n "$lt_option_debug"; then
      $ECHO "test_CTA:test_CTA:${LINENO}: newargv[0]: $progdir/$program" 1>&2
      func_lt_dump_args ${1+"$@"} 1>&2
    fi
    exec "$progdir/$program" ${1+"$@"}

    $ECHO "$0: cannot exec $program $*" 1>&2
    exit 1
}
...
```

It is now sufficient to replace exec by gdb, i.e.

```
$ nano test_CTA
...
# Core function for launching the target application
func_exec_program_core ()
{

    if test -n "$lt_option_debug"; then
      $ECHO "test_CTA:test_CTA:${LINENO}: newargv[0]: $progdir/$program" 1>&2
      func_lt_dump_args ${1+"$@"} 1>&2
    fi
    gdb "$progdir/$program" ${1+"$@"}

    $ECHO "$0: cannot exec $program $*" 1>&2
    exit 1
}
...
```

Now, invoking test_CTA will launch the debugger:

```
$ ./test_CTA
GNU gdb 6.3.50-20050815 (Apple version gdb-1515) (Sat Jan 15 08:33:48 UTC 2011)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "x86_64-apple-darwin"...Reading symbols for shared libraries ....... done

(gdb) run
Starting program: /Users/jurgen/git/gammalib/test/.libs/test_CTA
Reading symbols for shared libraries .++++++. done

****************************************
* CTA instrument specific class testing *
****************************************
Test response: .. ok
Test effective area: .. ok
Test PSF: ........ ok
Test integrated PSF: ..... ok
Test diffuse IRF: .. ok
Test diffuse IRF integration: .. ok
Test unbinned observations: ...... ok
Test binned observation: ... ok
Test unbinned optimizer: ......................... ok
Test binned optimizer: ......................... ok

Program exited normally.
(gdb) quit
./test_CTA: cannot exec test_CTA
```

Debugging Python unit tests

Have a look e.g. at test_GSky.py on how to write Python unit tests in gammalib.

Note that when you run your unit test via test_python.py, your Python code is actually run in a way that makes it impossible to debug your code. print statements will not print to the console and Python exceptions will abort your test, but not show up on the console or the GPython.xml unit test log file.

To write Python unit tests, you should first develop them in independent scripts or in the IPython console or notebook, and only when they are debugged, copy them into the gammalib test file, and add the GPythonTestSuite assert statements at the end of your test.

One possibility to debug Python in general is to add this line before the code you want to debug:

```
import IPython; IPython.embed()
```

This will drop you in an IPython interactive session, with the state (stack, variables) as it is in your Python script, and you can inspect variables or copy & paste the following statements from your script to see what is going on.

Installing GammaLib

If you would like to use GammaLib e.g. from standalone C++ programs or Python scripts or from ctools,
the procedure described above will not work.

You have to execute the additional make install step and source the gammalib-init.sh setup file so that you'll actually use the new version of GammaLib:

```
$ export GAMMALIB=<wherever you want>
$ ./configure --prefix=$GAMMALIB
$ make install
$ source $GAMMALIB/bin/gammalib-init.sh
```

If you are unsure which version of GammaLib you are using you can use $GAMMALIB or pkg-config to find out:

```
$ echo $GAMMALIB
$ ls -lh $GAMMALIB/lib/libgamma.*
$ pkg-config --libs gammalib
```

Look at the modification date of the libgamma.so file (libgamma.dylib on Mac) to check if it's the one that contains your latest changes.

Profiling GammaLib

To be written.

Adding a class to GammmaLib

To be written.

Adding an instrument to GammaLib

To be written.

Files

configure.out	10.2 KB	10/12/2012	Knödlseder Jürgen