EUROPEAN SOUTHERN OBSERVATORY

Organisation Européenne pour des Recherches Astronomiques dans l'Hémisphère Austral

Europäische Organisation für astronomische Forschung in der südlichen Hemisphäre

VERY LARGE TELESCOPE

Common Pipeline Library User Manual

VLT–MAN–ESO–19500–2720

Issue 1.0

Date 2003–12–15

	Name	Date	Signature
Prepared:	CPL Project Team	2003-12-15	
Approved:	M. Peron		
Released:	P. Quinn		

This page was intentionally left blank

Change record

Issue/Rev.	Date	Section/Parag. affected	Reason/Initiation/Documents/Remarks
1.0	15/12/2003	All	First version

This page was intentionally left blank

Contents

1 Introduction

1.1 The *Common Pipeline Library*

The *Common Pipeline Library* (CPL) consists of a set of C libraries, which have been developed to standardise the way VLT instrument pipelines are built, to shorten their development cycle and to ease their maintenance. The *Common Pipeline Library* was not designed as a general purpose image processing library, but rather to address two primary requirements. The first of these was to provide an interface to the VLT pipeline runtime-environment. The second was to provide a software kit of medium-level tools, which allows astronomical data-reduction tasks to be built rapidly.

The *Common Pipeline Library* provides:

- many useful data types (property lists, images, tables, ...),
- string and file utilities,
- medium-level data access methods (a simple data abstraction layer),
- image and signal processing capabilities,
- standard implementations of commonly used data reduction tasks,
- support for dynamic loading of recipe modules, and,
- standardised application interfaces for pipeline recipes.

Despite the bias towards instrument pipeline development, the library core provides a variety of general-purpose image and signal-processing functions. Thus, it also serves well as a basis for any generic data-handling package.

1.2 Future work

This first release of the *Common Pipeline Library* does not meet all the requirements for a data reduction library. However it establishes support for an extended set of data objects and methods that represent a solid foundation for further development of the CPL toward a fairly complete library.

Three major areas of growth are foreseen for the next release: further generalisation of basic data types, definition of a set of astronomy-oriented utility functions and implementation of high-level data reduction methods.

Among the possible generalisations of the CPL basic data types is the 3D extension of the CPL table component; currently a table element can just be a scalar or a character string. The CPL table component implementation will be extended to support numerical arrays and CPL images as column elements. More sophisticated methods for signal processing will also be added to any CPL basic component as they will be needed in the development of future pipelines.

A set of astronomical utility functions will also be created, to support spherical coordinate transformations, date and time conversions, precession, atmospheric extinction determination and other common operations in astronomy. It is considered especially desirable to support celestial WCS on CPL image frames, their determination on the basis of identified stars, and any related application such as image alignment and resampling.

Finally, a standardisation of the most common calibration steps and removal of instrument signature would be offered. Of course the data reduction system developer may freely define any specific procedure to support bias subtraction, flat fielding, wavelength calibration, instrument response linearisation, cosmic ray removal, object detection, bad pixel determination, etc., for any particular instrument. Yet, standard implementations of such tasks will be made available in the CPL for the most common detectors used for the VLT.

1.3 Acknowledgements

In June 2001, N. Devillard and R. Palsa first proposed a common software library in order to ease and accelerate the development efforts for the different VLT instrument pipelines. This software library, called *Common Pipeline Library* (CPL), would essentially be built up from already existing code. In particular, the *Eclipse* library (used for ISAAC and NACO pipelines) and concepts of the VIMOS data reduction software would be the main pillars of the CPL software.

In September 2001, M. Peron formed a CPL project team, consisting of N. Devillard and Y. Jung (working for ISAAC, NAOS/CONICA), together with R. Palsa and C. Izzo (working for VIMOS, FORS1/2), as well as P. Ballester and C. Sabet from the VLTI pipeline project. K. Banse served as mediator and chairman. In September 2002, N. Devillard left the CPL project and M. Kiesgen became a member of the team by the end of that year. Finally, L. Lundin, A. Modigliani and D. J. McKay joined the CPL team in the course of the year 2003. A preliminary version of the CPL was released in May 2002. Building on this basic version, the first official release of the CPL was made available to the public by ESO in December 2003.

1.4 Reference documents

[1] Data Flow for VLT instruments Requirement Specification VLT-SPE-ESO-19000-1618
[2] DFS Pipeline & Quality Control – User Manual VLT-MAN-ESO-19500-1619
[3] ESO DICB – Data Interface Control Document GEN-SPE-ESO-00000-0794
[4] Common Pipeline Library Reference Manual
[5] Recommended C Style and Coding Standards

1.5 Abbreviations and acronyms

CONICA	COudé Near Infrared Camera Array
CPL	Common Pipeline Library
DHS	Data Handling Server
DFS	Data Flow System
DO	Data Organiser
DRS	Data Reduction System
ESO	European Southern Observatory
ESO–MIDAS	ESO's Munich Image Data Analysis System
FORS	FOcal Reducer/low dispersion Spectrograph
FTP	File Transfer Protocol
ISAAC	Infrared Spectrometer And Array Camera
GNU	GNU's Not Unix!
LSS	Long Slit Spectroscopy
MOS	Multi Object Spectroscopy
NAOS	Nasmyth Adaptive Optics System
PDRM	Pluggable Data Reduction Module
RB	Reduction Block
RBS	Reduction Block Scheduler
SDK	Software Development Kit
UT	Unit Telescope
VIMOS	VIsible Multi-Object Spectrograph
VLT	Very Large Telescope
VLTI	Very Large Telescope Interferometer

2 Installation

This chapter gives generic instructions on how to obtain, build and install the *Common Pipeline Library*. Even if this chapter is kept as up-to-date as much as possible, it may not be fully applicable to a particular release. This might especially happen for patch releases. You are therefore advised to read the installation instructions delivered with the *Common Pipeline Library* distribution. These release-specific instructions can be found in the file `README` located in the top-level directory of the unpacked *Common Pipeline Library* source tree. The supported platforms are listed in Section 2.1. It is recommended that you read through Section 2.2.3 before you commence the installation procedure.

2.1 Supported platforms

The utilisation of the GNU build tools should allow you to build and install the *Common Pipeline Library* on a variety of UNIX platforms. The goal is to support the following target platforms:

- HP-UX 11.00 or later,
- Sun Solaris 2.8 or later,
- Linux (glibc 2.1 or later),
- DEC/Alpha (OSF/1 or Tru64),
- AIX, and,
- BSD compatibles.

However, only the VLT target platforms and operating systems, HP-UX 11, Solaris 2.8 and Linux (glibc 2.1 or later), are officially supported at this stage.

2.2 Building the CPL from the source distribution

This section shows how to obtain, build and install the *Common Pipeline Library* on your system from the official source distribution.

2.2.1 Requirements

To compile and install the *Common Pipeline Library* you need:

- **qfits** 4.3.5 or later,
- an ANSI/ISO-C99 compliant C compiler (preferably `gcc` 3.2 or later),
- the GNU `gzip` data compression program,
- a version of the `tar` file-archiving program, and,
- the GNU `make` utility.

2.2.2 Downloading the CPL source distribution

You may always obtain the latest release of the *Common Pipeline Library* sources from the ESO FTP server. To download the source distribution, point your browser to:

```
ftp://ftp.eso.org/pub/cpl
```

The CPL sources are distributed as a gzipped tar archive named in the format `cpl-X.Y.Z.tar.gz`, where `X` and `Y` are the major and minor release numbers, and `Z` denotes the patch level (which might be missing if no patch has been released).

In addition, the *Common Pipeline Library* depends on the **qfits** library (see section 2.2.1). The **qfits** distribution is available from the official download page at:

```
http://www.eso.org/projects/aot/qfits
```

To build the *Common Pipeline Library*, at least the static **qfits** library must be available on your system. If the shared object library is also present it will be used at run-time. Using the shared object library has the advantage that **qfits** can be updated, if necessary, without the need to re-compile the CPL afterwards.

2.2.3 Compiling the *Common Pipeline Library*

It is recommended that you completely read through this section before you actually begin with the installation.

1. First, if an appropriate version of **qfits** (c.f. section 2.2.1) does not already exist on your system, compile and install the **qfits** libraries. For detailed instructions on how to install the **qfits** libraries please refer to the **qfits** documentation.

   Typically, for an installation into the default directory `/opt/qfits` (you might need *root* privileges to do this) you must execute:

   ```
   $ zcat -d qfits-X.Y.Z.tar.gz | tar -xvf -
   $ cd qfits
   $ ./configure --enable-shared
   $ make static dynamic
   $ make install
   ```

   The following assumes that **qfits** is installed in `/opt/qfits`.

2. Unpack the CPL sources in a directory of your choice using

   ```
   $ zcat -d cpl-X.Y.Z.tar.gz | tar -xf -
   ```

   at the system prompt. This will create a directory called `cpl-X.Y.Z` containing the source tree.

3. Before running the configuration script it is recommended that you add some variables to your environment.

   The environment variable `QFITSDIR` tells the configuration script where the **qfits** libraries and header files can be found. Actually, this variable needs to be defined only if **qfits** has not been installed in the default directory `/opt/qfits` or any of the system's standard directories. The environment variable `CPLDIR` determines the installation prefix for the CPL. The default is `/usr/local` and usually the installation must be done as *root*.

   It is not mandatory to have the variables `CPLDIR` and `QFITSDIR` defined since you may pass the installation prefixes as command line options to the configuration script (c.f. 4). But packages depending on the CPL might look for these definitions at build time (see Section 3.3 for instance), so that it is simply convenient to have them defined as part of your environment. In the following, it is assumed that both `CPLDIR` and `QFITSDIR` are set correctly.

   Please note that assigning the default installation prefixes to the environment variables in the example below is just for demonstration purposes. In principle, they could be set to any directory for which you have write access with one exception: it is not recommended that you install the CPL into its own source tree.

   If your shell is the *Bourne* or a compatible shell (*i.e. sh*, *bash*, *ksh*, *zsh*, etc.) you should add:

   ```
   QFITSDIR=/opt/qfits
   CPLDIR=/usr/local
   PATH=$CPLDIR/bin:$PATH
   LD_LIBRARY_PATH=$CPLDIR/lib:$QFITSDIR/lib:$LD_LIBRARY_PATH
   export CPLDIR QFITSDIR LD_LIBRARY_PATH
   ```

   to the file `.profile` (or `.bashrc` if you are using `bash`). If you are using the C-shell (*i.e. csh* or *tcsh*) the commands above translate into:

   ```
   setenv QFITSDIR /opt/qfits
   setenv CPLDIR /usr/local
   setenv PATH $CPLDIR/bin:$PATH
   setenv LD_LIBRARY_PATH \
          $CPLDIR/lib:$QFITSDIR/lib:$LD_LIBRARY_PATH
   ```

   and should be added to the C-shell startup file `.cshrc`.

   The variable `LD_LIBRARY_PATH` is the dynamic linker's search path and allows an application to find the CPL libraries at run-time if they are not installed in one of the system's standard directories. Please note that the name of this variable may depend on the platform on which you are working. The name `LD_LIBRARY_PATH` is used on Linux and Solaris platforms whereas on an HP-UX system it is called `SHLIBS_PATH`. For details please refer to the documentation of your system; the dynamic linker's manual pages are a good starting point.

   To activate these settings you may either logout and login again, source the startup script manually. Alternatively, you may use the command line options of the configuration script, as described in 4. Note that if you are going to install dependent packages you might have to repeat these command line options for each of these packages, if the variables `CPLDIR` and `QFITSDIR` are not set.

4. To compile and install the CPL on your system run the following sequence of commands:

```
$ cd cpl-X.Y.Z
$ ./configure
$ make
$ make install
$ make install-html
```

Before installing the CPL on your system you may want to verify that the CPL was built correctly. This can be done by running the command `make check` before executing `make install`. This will build and run some test cases and it will output a short summary of the test results at the end.

The last command, `make install-html`, is optional and installs the *Common Pipeline Library* On-Line Reference Manual into the directory `$CPLDIR/share/doc/cpl/html`. The on-line documentation for the C Extension Library may be found in `$CPLDIR/share/doc/cext/html`.

The `configure` script provides a variety of command-line options to customise the CPL installation. The list of available options can be obtained by running `./configure --help` in the top-level directory of the source tree. Using a command line option always takes precedence over any previously set environment variable. In particular, the variables `CPLDIR` and `QFITSDIR` are overridden by the options `--prefix` and `--with-qfits` respectively.

At this point, the installation of the *Common Pipeline Library* is complete and you can start using it. If the installation did complete successfully, you may also safely delete the whole source tree to save disk space, as it is no longer needed.

If the CPL has been installed into one of the system's standard directories, the dynamic linker search path does not need to be modified, as these directories are searched by default. But on Linux systems, it might be necessary to update the dynamic loader's cache by executing the command `ldconfig` as *root* at the system prompt.

3 Software development with the CPL

This section gives a short overview on how the *Common Pipeline Library* can be used to develop your own software, either simple applications, just using the facilities provided by the CPL libraries, or *Pluggable Data Reduction Modules* (PDRM), to be used as part of one of ESO's VLT instrument pipelines.

3.1 Getting started

In this document we assume that you know the ANSI C programming language, your C compiler and that you are also familiar with the GNU *make* utility.

Before you start coding it is recommended that you, at least, skim through this manual to get a short overview of the components provided by the CPL. In the following chapters you will also find code snippets which demonstrate the typical usage of the various components. Two small examples illustrating the two different kinds of CPL 'applications' can be found in the Sections 3.4 and 3.5.

After making yourself familiar with main CPL components and concepts, you can start working on your project by having a look at the CPL on-line reference manual to get in depth knowledge of the CPL components you want to use.

3.2 Using the *Common Pipeline Library* in your project

If you want to use the CPL, you need to know where the header files and the libraries are installed. By default, the CPL header files and libraries can be found in the subdirectories `include` and `lib` of the root directory of your CPL installation, but the actual location might be different depending on the configuration options used at build time.

In the following, it is assumed that the CPL has been installed in its default location `/usr/local`, so that the header files are located in `/usr/local/include` and the libraries can be found in `/usr/local/lib`.

Alternatively, the GNU build tools *autoconf*, *automake* and *libtool* may be used. In general, this is the recommended way to compile and link your application. Especially if you are going to develop CPL plugins, the use of the GNU build tools makes dealing with shared object libraries for different platforms a lot easier. Comprehensive information on the GNU build tool can be found via `http://www.gnu.org`.

The CPL provides support for the GNU build tools by providing a small collection of autoconf macros in the two macro archives `cpl.m4` and `eso.m4`. These archives contain, among others, macros to locate the CPL header files and libraries on your system and to setup the appropriate `Makefile` symbols needed to compile and link a CPL application. You can find them in the CPL source tree in the subdirectories `m4macros` and `libcext/m4macros`. To use them copy the two files to the source tree of your own project so that they can be found by the *aclocal* tool, which is part of the GNU *automake* package.

If you are going to develop a fully-fledged VLT instrument pipeline, the use of the GNU build tools is not only recommended, but required. An appropriate CPL SDK containing the necessary tools and a template directory tree will be available on request.

3.3 Linking your application with the CPL

The CPL libraries *libcplui*, *libcplbase* and *libcplcore*, together with *libcext* and the *libqfits* library, form a hierarchy, *i.e.* there are inter-library dependencies, of which you need to be aware, when linking your application. Figure 1 shows the library dependencies of a CPL application using functionalities from all the CPL libraries.

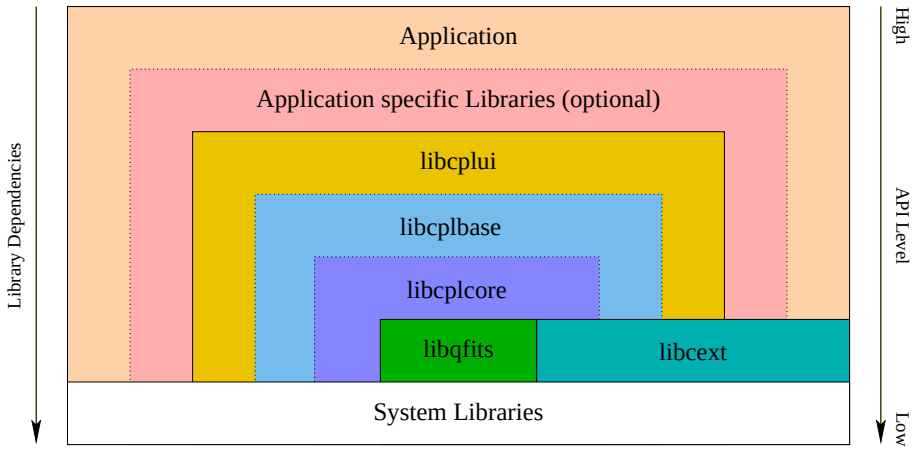

Figure 1: Library dependencies of a CPL application

For an application as shown in Figure 1, the linker command would look like the following, with the trailing ellipsis being a placeholder for any system libraries that are also used:

```
$ gcc -o myapplication myapplication.o -lmylibrary \
> -L$CPLDIR/lib -lcplui -lcplbase -lcplcore -lcext \
> -L$QFITSDIR -lqfits ...
```

The order in which the libraries are linked matters and is determined by the inter-library dependencies. This implies that the order of linking for the two libraries *libcext* and *libqfits* does not matter in the above example. Actually, these two libraries may even be skipped, since the CPL library *libcplcore* usually includes these dependencies, so that running the command

```
$ gcc -o myapplication myapplication.o -lmylibrary \
> -L$CPLDIR/lib -lcplui -lcplbase -lcplcore ...
```

should be sufficient.

An application programmer is free to choose which CPL facilities he or she wishes to use and therefore needs to link only with the libraries upon which the highest-level library used depends. Therefore, for an application which uses only components from *libcplcore*, the above linker command would become:

```
$ gcc -o myapplication myapplication.o -lmylibrary \
> -L$CPLDIR/lib -lcplcore -lcext -L$QFITSDIR -lqfits ...
```

3.4 Writing a simple *Common Pipeline Library* application

The CPL libraries can be used as any other library on your system to write applications. This section provides you with a simple example of how to do this; CPL's "*Hello, world!*" program:

```
#include <cpl_init.h>
#include <cpl_messaging.h>

int main()
{

    cpl_init();

    cpl_msg_start();
    cpl_msg_info("hello()", "Hello, world!");
    cpl_msg_stop();

    return 0;

}
```

Compiling this program and running it at the system prompt produces the output:

```
$ ./hello
[ INFO ] Hello, world!
```

Line-by-line Walkthrough

The first line

```
#include <cpl_init.h>
```

includes the prototype of the CPL initialisation function and the second line

```
#include <cpl_messaging.h>
```

includes the services of the CPL messaging component. As with every C-program, a CPL application has to start with the usual definition of the *main*-function:

```
int main()
{
```

The first function call

```
cpl_init();
```

initialises the CPL. In particular, the library's memory management system is initialised. The function `cpl_init()` **must** be called before any other CPL function is called!

Now the application can start doing the real work. After the library has been initialised, the CPL messaging system is started by calling

```
cpl_msg_start();
```

The function call

```
cpl_msg_info("hello()", "Hello, world!");
```

writes the well-known message to the terminal, with a prefix indicating the message severity. The first argument, the string "`hello()`", is the component tag and indicates the program, module or function which emits the message. The component tag is not printed by default and therefore does not appear on the screen. The last function call in this example

```
cpl_msg_stop();
```

shuts down the CPL messaging system.

The program ends with a successful return from `main()`:

```
   return 0;
}
```

The previous example shows the basic layout of any CPL application. After the library initialisation and the setup of the messaging system your application can use all the facilities provided by the CPL.

For further details on the messaging component please refer to Section 5.2.6 and the CPL reference manual [4].

3.5 How to implement a Pluggable Data Reduction Module

This section shows how a simple data reduction task, namely doing basic arithmetic with two images, can be implemented using the CPL plugin interface.

What is a plugin

A *plugin* is a unit of code that can be incorporated into a parent application at run-time. Unlike a static or dynamic library, the details of the plugin's existence do not need to be known by the parent application when

it is built and vice versa. As such, plugins are extremely useful for pipeline-management software or GUIs, where the developers may wish to modify parts of the pipeline code, without necessarily restarting the parent application (let alone recompiling it).

In a way, this is similar to spawning a child process (although plugins are, in general, executed synchronously). However, the child-process method then needs to take into consideration communication with the parent application, which means the definition of, and strict conformance to, an interface specification, which is then difficult to check outside the run-time environment. It also means that the child process needs to implement some interprocess communication methods.

In comparison, a plugin implements its interface simply through the provision of four function calls, that are expected by the CPL plugin interface in the parent application. The parent application does not need to know about the plugin's existence at compile time, but can learn about the plugin's existence via user input or a configuration file, during normal execution. It can then query the existence of the plugin, and again handle the case where the plugin is not available in a graceful manner.

If the plugin is available, then the code within it may be invoked by this standard interface. Of course, the downside is that, unlike a completely separate child process, the plugin is executed within the address space of the parent application, which means that fatal errors (e.g. segmentation fault) will take down both components, unless the appropriate provisions are made.

What is a PDRM

A *Pluggable Data Reduction Module* (PDRM) is just a specialised type of plugin, suitable for implementing a data reduction task, *i.e.* a *recipe*. In other words, if a *recipe*, is implemented using the CPL plugin interface, it is called a *Pluggable Data Reduction Module*.

This section demonstrates how easy it is to implement such a *Pluggable Data Reduction Module*. It is easy, because a plugin developer does not need to know how the input for the data reduction task is created. He or she can expect that the complete information the data reduction task needs is available when it executes. All the "nitty-gritty" details of command line parsing, file management, etc., are left to the application using the plugin.

What is needed

To implement a PDRM, four functions have to be implemented which are used by the application to obtain some information about the plugin, to initialise, execute and "clean it up". In addition, one or more functions doing the real work are needed too.

An Example

The example shown below describes a PDRM which supports basic arithmetic with images. It will provide one option, for selecting the arithmetic operation to be executed.

The first function to implement is the one that the application will call initially in order to obtain the necessary information about the plugin. This function is described as part of the plugin interface, *i.e.* the function's prototype and its name are defined by the interface but the function needs to be re-implemented by each plugin.

This is the only function which needs to be exported by the PDRM, *i.e.* this is the only function which must not be declared `static` in the module's source file.

The function is called `cpl_plugin_get_info`, returns an `int`, takes a pointer to `cpl_pluginlist` as its only argument and it can be implemented either using the public interface of the plugin directly or the provided convenience function. An implementation, completely ignoring error handling to keep it simple, would look like:

```
#include <cpl_memory.h>
#include <cpl_recipe.h>
#include <cpl_plugininfo.h>

#define MY_PLUGIN_VERSION  1

/* Plugin detailed description */

static const char *
myplugin_help = "The plugin adds, subtracts, multiplies or divides "
                "two images depending on the operation choosen by the "
                "parameter ‘operation’.";


int myplugin_create(cpl_plugin *);
int myplugin_exec(cpl_plugin *);
int myplugin_destroy(cpl_plugin *);


int
cpl_plugin_get_info(cpl_pluginlist *list)
{

    cpl_recipe *recipe = cpl_calloc(1, sizeof *recipe);
    cpl_plugin *plugin = (cpl_plugin *)recipe;


    cpl_plugin_init(plugin,
                    CPL_PLUGIN_API,
                    MY_PLUGIN_VERSION,
                    "myplugin",
                    "Do basic arithmetic on two images",
                    myplugin_help,
                    "Gill Bates",
                    "gbates@macrohard.com",
                    "GPL",
                    myplugin_create,
                    myplugin_exec,
                    myplugin_destroy);

    cpl_plugin_list_append(list, plugin);

    return 0;
```

```
}
```

The first three lines include the definitions of the CPL memory services, the `cpl_recipe`, `cpl_plugin` and `cpl_pluginlist` types.

The symbol `MY_PLUGIN_VERSION` is defined to be the recipe's version number and the static variable `myplugin_help` is assigned to the recipe's detailed description. This is followed by the forward declarations of the three remaining functions which must be implemented to create, execute and destroy the recipe.

The function `cpl_plugin_get_info` is implemented as follows. First, memory to hold the recipe object is allocated. The subsequent cast of the variable `recipe`, which is a pointer to `cpl_recipe`, into a pointer to `cpl_plugin` is possible because the class `cpl_recipe` is a subclass of `cpl_plugin` (see the ISO-C standard ISO/IEC:9899:1999(E) 6.7.2.1 for details).

The `cpl_plugin` part of the recipe object is then initialised with the version of the `cpl_plugin` class implementation, the recipe's version, the name of this recipe plugin, a short description of its purpose, a longer help text and license information. The last three arguments passed in the call to `cpl_plugin_init` are the functions the application will use to initialise, execute and destroy the recipe plugin. Their implementations are discussed below.

As a last step, the plugin is appended to the list of plugins. This list must be provided by the application calling `cpl_plugin_init`. At this point, the creation of the recipe plugin with all necessary information is completed and the function returns successfully.

What is left to be done is the implementation of the initialisation, execution and cleanup functions. In the beginning, it was mentioned that our example should be configurable insofar, that a user may select the arithmetic operation to be performed. It is the duty of the PDRM to provide the information about any options it accepts to an application which uses the PDRM. In our example, we need to define our arithmetic operator option. The correct place to do this is the PDRM's initialiser function. The created parameter(s) are stored in a parameter list, which can be queried and updated by the calling application. These configuration parameters may, for instance, be mapped into command line options by the calling application. Since the recipe configuration is created during the plugin's initialisation, it has to be destroyed in the end, namely, in the plugin's cleanup handler. A typical implementation of these two functions looks like:

```
static int
myplugin_create(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;
    cpl_parameter *p;


    recipe->parameters = cpl_parlist_new();

    p = cpl_parameter_enum_new("myplugin.operation",
                               CPL_TYPE_STRING,
                               "Arithmetic operation to apply.",
                               "myplugin",
```

```
                                "add", 4,
                                "add", "subtract", "multiply", "divide");
    cpl_parameter_set_alias(p, "op", NULL, NULL);
    cpl_parlist_append(recipe->parameters, p);

    return 0;

}


static int
myplugin_destroy(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;


    cpl_parlist_delete(recipe->parameters);

    return 0;

}
```

In the very beginning, both functions must convert the plugin which has been passed to them from a pointer to `cpl_plugin` into a pointer to `cpl_recipe` to get access to the additional members that the `cpl_recipe` class provides. This cast operation is safe since the plugin has been explicitly instantiated as a `cpl_recipe` in the `cpl_plugin_get_info` function, that was called initially.

The recipe subclass has two additional members compared to its superclass, the generic plugin. These two data members are the list of recipe configuration parameters and the set of input data frames which it should process. The list of accepted configuration options is created by the recipe while the set of input frames must be filled in by the calling application.

In the remainder of the initialisation function, a parameter list and an enumeration parameter is created (please refer to [4] for the technical details on how to create the various kind of parameters). The created parameter will allow the selection of the arithmetic operations supported by the recipe. Changing its value, via the calling application's user interface, will configure the PDRM using the requested operator during its execution. For the user's convenience, a short alias name for the parameter is provided which may be used by an application instead of, or in addition to, the parameter's fully qualified name. Finally, the parameter is appended to the parameter list. The only operation which is necessary in the cleanup handler is the one required to destroy the parameter list and all its contents, therefore its implementation is straight forward.

The last interface function which is needed is the function to execute the recipe. Again the implementation is straight forward, assuming that the actual processing function `my_image_arithmetics` does all the work.

```
static int
myplugin_exec(cpl_plugin *plugin)
{
```

```
    cpl_recipe *recipe = (cpl_recipe *)plugin;

    return my_image_arithmetics(recipe->parameters, recipe->frames);

}
```

The implementation of the processing function `my_image_arithmetics` is left to the reader as an exercise.

The three functions initialising, executing and destroying the recipe plugin are defined as `static` functions. There is no need to make them publicly available because they are exported by the plugin interface itself and they are only called through this interface.

As mentioned before, the example does not implement any error handling. For the three handler functions and the function to obtain the plugin information it is required that they return `0` on success and a non-zero value to indicate an error.

The complete source code of the example can be found in appendix B. To try it, you should build a shared object library from the source and you must provide the actual processing function.

3.6 Current library limitations

- Currently the CPL does not provide the *libcplbase* library. It will be made available in future releases.
- The CPL libraries and also the **qfits** library are not thread-safe!

4 CPL general design features

4.1 OO approach

The CPL has been written in C, but following an object-oriented (OO) approach wherever it makes sense. Modules are built around a class, which comprises a `typedef` (usually a `struct`) and a list of associated methods to work on it.

For example, the image class is built like this:

```
/* Class definition */
typedef struct _cpl_image_ {
  ... CPL image attributes ...
} cpl_image ;

/* Associated methods */
cpl_image *cpl_image_new(...);
cpl_image *cpl_image_copy(...);
void   cpl_image_delete(...);
```

Understanding the library means parsing through the list of offered components and looking at the implemented methods. There are components for the handling of the data to process (images, cubes, pixel maps, tables, vectors, ...) and purely functional components to help programmers, such as the messaging and the error handling components.

'Data hiding' is used wherever applicable. Most objects remain opaque and are only manipulated through accessor functions. See the documentation for each component.

Polymorphism is hard to achieve in C, and is seldom used, if at all, in the CPL. The OO approach is limited here to defining objects with attributes and methods.

4.2 Portability

The CPL is intended to have a long service life and evolve in accordance with the needs of the VLT. To avoid locking the code to any particular platform, portability has been considered throughout the design of the CPL. Achieving portable code is done in the CPL through tools like `autoconf` and `automake` that try to catch all system dependencies and make them look the same to library users, ironing out any local peculiarity (*e.g.*, HP-UX lacks many standard tools or has them with different names). But this is not the end of the story. During development, we kept in mind all the basic portability rules and relied on the use of compiler options (like `-ansi`, `-pedantic-errors`, `-Wall`), and tools such as `lint`. The aim was that the CPL should be usable on any kind of POSIX-compatible system.

System-specific optimisations may be added later if they do not involve modifying any API in the code. If optimisations are introduced, they shall be resolved at compile-time and hidden from library users.

4.3 FITS I/O

The library used by the CPL to handle FITS I/O is **qfits**. This is a stand-alone library supporting all required FITS file accesses. It is described in its own document; see the **qfits** web page for more information:

`http://www.eso.org/projects/aot/qfits`

The **qfits** library makes use of the `xmemory` functions for any memory handling (see Section 4.4).

4.4 The `xmemory` memory model

`xmemory` stands for "Extended memory". It consists of a set of memory allocation/deallocation functions:

```
xmemory_malloc()
xmemory_calloc()
xmemory_realloc()
xmemory_free()
xmemory_strdup()
```

These functions are meant to replace the default standard library functions that control and handle all memory allocation in applications.

4.4.1 Advantages of using `xmemory`

By using `xmemory`, memory can be allocated past the normal hardware limitations of the machine, *i.e.*, more than 'RAM + swap' can be allocated. To achieve this, the `xmemory` system will create its own swap files as necessary.

Pointers allocated with `xmemory` will never be NULL. The `xmemory` system would `exit` the application if this should really happen. So, if this model is used, it can be assumed everywhere in the code that memory is always present in large quantities, making the handling of very large files not an issue, thus avoiding the need to split input data into smaller chunks for handling at the lowest level.

It is possible to check for memory leaks at any moment using the appropriate memory-report function.

4.4.2 Using `xmemory`

Inside the CPL, every memory allocation function (*e.g.*, `malloc()`) is redirected to its `xmemory` equivalent (`xmemory_malloc()`). This process is invisible, since the standard allocation function names are used in the code. Thus, developers should continue to use the normal system functions, but simply bear in mind that they have been overloaded by the statement:

```
#include "xmemory.h"
```

Outside the CPL, the situation is different. The memory allocated inside the CPL has to be deallocated using `xmemory` functions. This can be done either with the CPL objects destructor (*e.g.*, `cpl_image_delete()`) to deallocate CPL objects or with `cpl_free()` for normal arrays created by CPL functions.

You are free to use `xmemory` to allocate/deallocate your memory in your code with `cpl_malloc()`, `cpl_calloc()`, `cpl_realloc()` or `cpl_free()`.

The only rule is that all the memory allocated with the `xmemory` model must be deallocated with it.

The access to these memory functions is provided by the statement:

```
#include "cpl_memory.h"
```

Please refer to appendix A for a more detailed description on the `xmemory` model.

4.4.3 Documentation on `xmemory`

This model is completely described in its own design and implementation document, which can be found on the **qfits** web page (see section 4.3).

4.5 Error handling

Error handling in the CPL is done through the `cpl_error` component (see Section 5.2.7).

4.6 Library stability

The CPL group will strive to keep the API stable, in order to allow for an easier maintenance of the many VLT pipelines. New releases will mostly provide new functionality and bug fixes, but radical design changes will be avoided as much as possible.

4.7 Code conventions

The coding conventions adopted in the CPL are basically the ones described in Recommended C Style and Coding Standards [5]. Although the coding language used is ISO-C [ISO/IEC:9899:1999(E)], the CPL developers have adopted an object oriented approach. A series of objects are defined (image, table, etc.) in the library and methods are associated to them.

4.7.1 Objects

An object is a C *structure* that contains all the information needed to describe it. The objects chosen to populate the CPL have been designed to be as small as possible. All the attributes associated with an object are mandatory.

An image, for example is defined by an array of pixels, the image size in X and Y, the pixel type and a bad pixel map; nothing more.

Adding more fields that are used only for some particular purposes in particular cases is an open door to end up with huge objects in which we never know which attribute has been initialised/updated and which not.

If more complicated objects are needed, it is left to the developer to define his or her own local object composed of the basic objects and other additional parameters defined as attributes.

Each object has one *constructor* that allocates the memory it needs, and a *destructor* that deallocates it. The destruction of objects should *always* be done through its dedicated method.

4.7.2 Methods

Apart from the constructor and destructor, other methods are associated with an object. Basically, each function that does something with an object or that modifies an object will be considered as a method of this object.

Any method can create or modify an object. In the latter case, the modified object should be passed as the first parameter to the function. Of course, a method can also use a passed object without modifying it.

In the case of a failure, the input object shall *always* remain unchanged.

4.7.3 Functions

All functions shall be able to inform their caller about the success of their execution, either by returning an error code (CPL_ERROR_NONE in case of success, the proper error code otherwise) or by returning a conventional value (as a NULL pointer when a pointer is expected) and setting appropriately the error code (see section 5.2.7).

4.8 Naming conventions

First of all, each name in the CPL (file, function, object, variable, ...) is composed of words/abbreviations in lower case, separated by underscores.

Example: *my_variable* is valid, but *My_Variable* or *MyVariable* or *myvariable* are not.

Furthermore, each function name, file name or object name in the CPL has been prefixed with *cpl_*.

Example: the image object is called *cpl_image*.

Included library in the CPL respect the same convention. They are prefixed by an identifier that make it clear from where the object/function/file is derived. It is the case for **qfits** where the prefix *qfits_* is used.

Each method name begins with the name of the object with which it is associated.

Example: cpl_image_add().

The two associated *constructor* and *destructor* methods are always named in the same way: object_name_new() and object_name_delete().

Example: cpl_image_new_float() and cpl_image_delete().

5 The CPL components

5.1 Component libraries

The functionality of the CPL is provided by three component libraries, implementing the low-, medium- and high-level CPL interfaces respectively. This allows applications to be linked with only the parts of the *Common Pipeline Library* that are necessary.

The core library, **libcplcore**, provides the basic types like vectors, images and tables, as well as the basic signal and image processing functionalities. It also provides facilities for accessing data files, for error signalling, and a set of functions for displaying messages and maintaining log files. Standard implementations for instrument-independent data-reduction functions and functions for monitoring the data quality are provided by the **libcplbase** library. Finally, the **libcplui** library implements the high-level data types and utilities serving as an interface to the pipeline run-time environment.

For the low-level implementation of container data types (such as lists, or dictionaries), or utilities not available on every UNIX system, the CPL libraries themselves depend on a small C library *libcext* extending the standard C library. This library is provided as part of the CPL package.

For access to FITS data files, the CPL internally relies on the **qfits** FITS I/O library (see the **qfits** Reference Manual for details). Since the CPL provides high-level facilities to read and write data from/to a FITS file, direct calling of **qfits** functions is almost never needed, and should be limited to the definition of functions that load FITS data into internal objects, and functions that save internal data objects to FITS files.

5.2 Core objects in *libcplcore*

5.2.1 Images

A *cpl_image* is conceptually a 2-dimensional array of pixels with two main characteristics. Firstly, a *cpl_image* can be of several different types (currently supported are *double*, *float*, *int* and *binary* — *i.e.*, boolean — images). Secondly, each *cpl_image* can carry with it the knowledge of its own bad pixels, referred to as a bad pixel map.

All the CPL functions of type *cpl_image* (and only those) allocate a new *cpl_image*. Any allocated *cpl_image* must be deallocated using the `cpl_image_delete()` function.

The following operations can be performed through the *cpl_image* methods' interface:

- creating, loading from FITS files, saving to FITS files or deallocating images,
- copying images, converting images from one type to another or accessing image information,
- set or unset bad pixels in an image, count them, set the bad pixels from an ASCII file or from a binary image,
- logical or morphological operations on binary images, selection and labelise functions,
- basic image operations, normalisation, thresholding, averaging, collapsing, extraction or flipping,

- various statistical computations on images,
- linear, median or morphological filtering operations, and,
- generation of images with random uniform noise, or with gaussian functions.

The different image components are described in the following sections. For some of them (*e.g.*, `cpl_image` or `cpl_image_bpm`), the way the data are stored internally is described. This is just to give a better idea on what the CPL can do and how efficient it can be. But these internal structures should *never* be accessed directly; every developer must restrict himself to only use the accessor functions provided in the library. By doing so, you ensure that you do not need to change your code after any CPL update, as the internal structures may change from one release to the next.

1. **The image structure (`cpl_image` component)**

   An image comprises a size in x and y (in pixels), and a pointer to an array of pixels. The type field, and the fact that the pixels are defined as void, allows this structure to contain any of the supported image types (float, double, integer or even binary images).

   The image-processing functions provided in the CPL can handle any meaningful kind of image. A user would call the same function to filter a double or a float image.

   Moreover, it is possible to attach to any image the knowledge of its bad pixels with the badpixelmap field (stored as a *cpl_sparseimage* object). Again, any image processing function in the CPL takes this bad pixel map into account whenever one is defined.

   The implementation of the *cpl_image* structure looks like:

```
typedef struct _cpl_image_
{
    int                nx, ny;
    cpl_type           type;
    void              *pixels;
    cpl_sparseimage   *badpixelmap;
} cpl_image;
```

   The image pixel buffer is two-dimensional but stored in a 1-dimensional array of pixels for efficiency reasons. Pixels are numbered (like arrays in C) from 0 to $nx \cdot ny - 1$.

   Note that this pixel organisation does not pre-suppose any given orientation for the lines in the image. The CPL convention, like the FITS convention (and as opposed to most other image formats), numbers lines from bottom to top. However, this is not an issue for most image operators. The pixel in the i-th column and the j-th row (starting at the lower left corner, conventionally corresponding to column 1 and row 1) would be the pixel number $(i-1) + (j-1) * nx$ in the array (see Figure 2).

   Changing the value of `nx`, `ny`, `type` or `pixels` directly is likely to corrupt the data and generate unpredictable behaviour. As stated earlier, these fields should *never* be accessed directly. Accessor functions are provided for this purpose (see IO routines description).

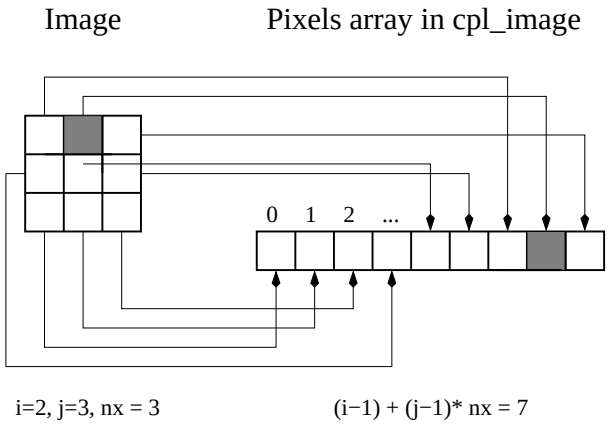

Figure 2: Pixel storage in the 1D data array

2. **The attached bad pixel map (`cpl_image_bpm` component)**

   The badpixelmap field in the *cpl_image* could have been a binary image in which the bad pixels would be tagged. But the number of bad pixels is usually a small fraction of the total number of pixels. So, to reduce the memory consumption, only the positions of the bad pixels are stored. The structure used is a *cpl_sparseimage* defined as:

```
typedef struct _cpl_sparseimage_ {
    int      nz;
    int     *indices;
} cpl_sparseimage;
```

   As for the *cpl_image* object, the fields should *never* be accessed directly, but with the provided accessor functions.

   The meaning of the *badpixelmap*-field is defined as:

   - *badpixelmap = NULL*: No knowledge of bad pixels.
   - $nz = -1$: No knowledge of bad pixels.
   - $nz = 0$: All pixels are known to be good.
   - $nz > 0$: At least nz pixels are known to be bad.
   - *indices != NULL* if $nz > 0$.

   When $nz > 0$, the indices array contains the nz positions of the bad pixels in the *cpl_image* pixels array in increasing order and in the range 0 to $nx \cdot ny - 1$ (see Figure 3).

   The `cpl/tests/cpl_image_bpm-test.c` file contains examples of `cpl_image_bpm` function usage.

3. **The image IO routines (`cpl_image_io` component)**

   There are two kind of functions that can be used to generate *cpl_image* objects from scratch or from a FITS file.

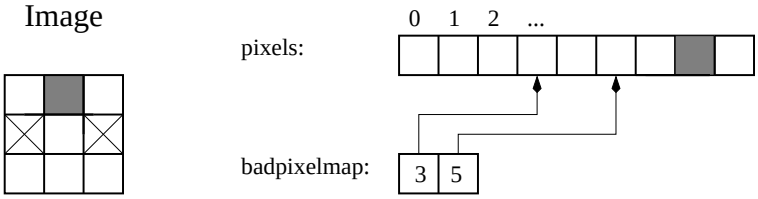

Figure 3: Bad pixel map information storage

The `cpl_image_new_xxx()` functions (where `xxx` is the required type among `int`, `float`, `double` or `binary`) will create new empty images (some with allocated data buffer, some without).

The `cpl_image_load()` will load an image from a FITS file. If you load an image from a FITS file, you have to specify which plane (you can store cubes in FITS files) in which extension, which type of image you require, and the function will give back to you the specified newly allocated *cpl_image*.

Examples:

```
cpl_image  *im1;
cpl_image  *im2;
cpl_matrix *kernel;

/*
 * Create a new image.
 * CREATES A NEWLY ALLOCATED OBJECT THAT MUST BE DESTROYED.
 */

im1 = cpl_image_new_float(1024, 512);

/* Define the kernel */
...

/*
 * Apply a median filter on im1.
 * CREATES A NEWLY ALLOCATED OBJECT THAT MUST BE DESTROYED.
 */

im2 = cpl_image_filter_median(im1, kernel);
cpl_matrix_delete(kernel);

/*
 * Subtract im2 from im1, a local operation.
 * DOES NOT CREATE ANY NEWLY ALLOCATED OBJECT.
 */

cpl_image_subtract_local(im1, im2);

/* Delete both images */

cpl_image_delete(im1);
```

```
cpl_image_delete(im2);
```

Please note that some *cpl_image* generation functions are provided in the *cpl_image_gen* component. These ones are mainly used in our testing facilities.

This component also provides the possibility to convert images to another type, to save images to a FITS file or to duplicate images. It also provides a series of accessor functions to retrieve the image size, type, number of bad pixels or a pointer to the data buffer.

The `cpl/tests/cpl_image_io-test.c` file contains examples of `cpl_image_io` function usage.

4. **The basic image operations (`cpl_image_basic` component)**

   This component offers the possibility to apply basic operations between images, including element-wise addition, subtraction, multiplication and division.

   Since all but unary operators may have image operands of different types we define the type of the result to be that of the first operand. This means that with the CPL, the addition or multiplication of two images of different types is non-commutative.

   The default image operator is of type *cpl_image* and allocates a new image for the result. Additionally, for some of the image operators, the CPL offers an equivalent assignment operator, which is named by appending *_local* to the function name, *e.g.*, *cpl_image_add_local*. These functions store their result in their first operand and are of type *cpl_error_code*.

   We define the result of an arithmetic operation on two pixels of which one or both are bad to be a bad pixel.

   The resulting bad pixel map of an element-wise-operation on two images is therefore the union of the bad pixel maps of the two operands. See Figure 4.

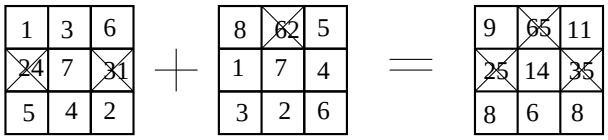

Figure 4: Bad pixel map handling in basic images operations

   For performance reasons, the operations are actually computed on all pixels (including any bad ones).

   Functions between an image and a scalar variable are also offered (addition, subtraction, multiplication, division, logarithm and exponential). In this case, the bad pixel map and the image type remain unchanged.

   Extraction, rotation, thresholding, collapsing and normalisation functions are also available. The handling of the bad pixels in these functions is intuitive.

   In the normalisation, the scaling factor is computed using the CPL image statistics functions which ignores the bad pixels.

   In the collapsing function, bad pixels are ignored in the flux summation (normal behaviour of the statistics function), with a result that has a bad pixel only in the rare case where all pixels along the collapsing direction are bad (see Figure 5).

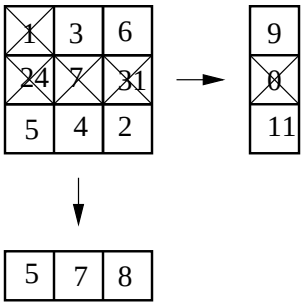

Figure 5: Bad pixel map handling in the collapsing function

The `cpl/tests/cpl_image_basic-test.c` file contains examples of `cpl_image_basic` function usage.

5. **Statistics on images (`cpl_image_stats` component)**

   Several functions providing various statistics on *cpl_image* objects are offered: the value and position of the minimum and maximum pixels, the mean, standard deviation, median, absolute flux and flux in the image or just in a rectangular part of the image. Real-valued statistical functions are implemented as type *double* regardless of the type of the input image. The statistics ignore bad pixels as shown in Figure 6.

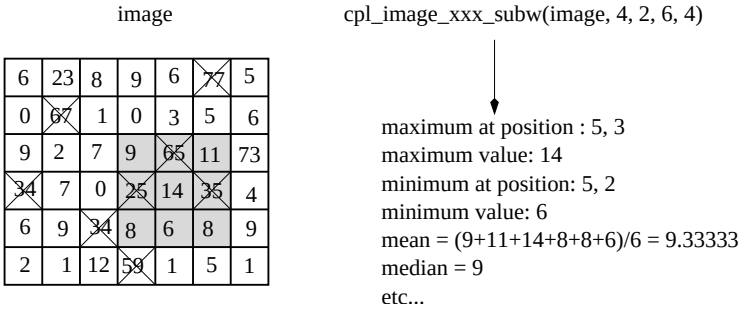

Figure 6: Bad pixel map handling in statistics computations

The `cpl/tests/cpl_image_stats-test.c` file contains examples of `cpl_image_stats` function usage.

6. **The image filtering functions (`cpl_image_filter` component)**

   This component offers linear filtering, morphological filtering, median filtering and standard deviation filtering.

   Without a separate handling of bad pixels, filtering involving a bad pixel will typically corrupt the neighbouring pixels as shown in Figure 7.

   In filtering it is therefore a significant improvement to be able to identify bad pixels and handle them properly. In the CPL, the filter functions simply ignore the bad pixels, and use only the good ones in the neighbourhood to compute the new value.

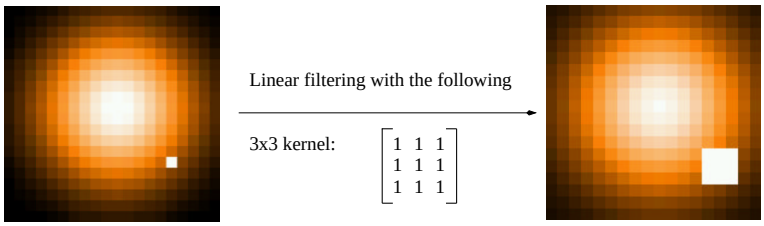

Figure 7: Filtering without bad pixels handling

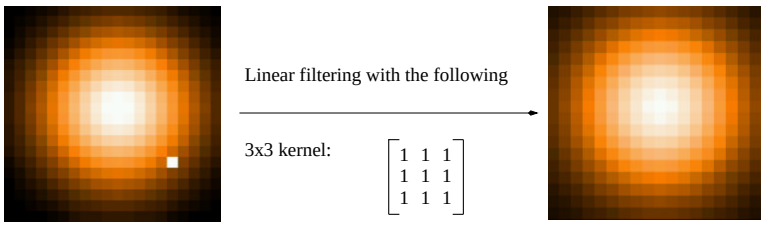

Figure 8: Filtering with the pixel (16, 6) tagged as bad

Figure 8 shows the result obtained when the bad pixel is correctly tagged.

This example shows that it is very important to flag the bad pixels as such; the neighbours are not affected by the filtering, and the bad pixel itself can be recomputed using the good neighbours. The only case where a bad pixel stays bad in the filtered image is when it only has bad pixels as neighbours.

Please note that the borders of the filtered image are set to 0 in the filtered image without being flagged as bad pixels.

The `cpl/tests/cpl_image_filter-test.c` file contains examples of `cpl_image_filter` function usage.

7. **The use of binary images (`cpl_image_binary` component)**

   A binary image is an image in which pixels can only have two different values. This type of image is widely used (and very useful) in image processing for object or edge detection. Furthermore, binary images are very useful during operations that take bad pixel maps into account.

   This particular type of *cpl_image* comes with the basic morphological operations like erosion, dilation, closing and opening, and also the logical operations like `and`, `or`, `not` and `xor`.

   A basic thresholding function to “binarise” a *float* or *double* image is provided. Figure 9 illustrates its effect on an example, where the threshold is computed with the *cpl_image_stats* functions on the input image.

   Some simple morphological operation can be applied to the binary image to make one connected object out of each detected star as shown in Figure 10. The operation applied here is a closing (erosion + dilation).

   Once the different objects are connected, we can apply a labelisation on the image to differentiate them automatically (see Figure 11). The binary image is transformed into an integer image where the non-

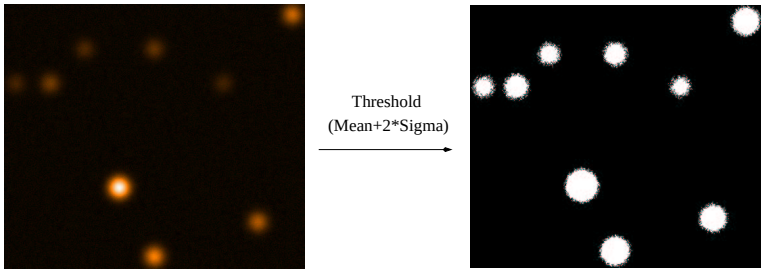

Figure 9: Use of thresholding to binarise an image

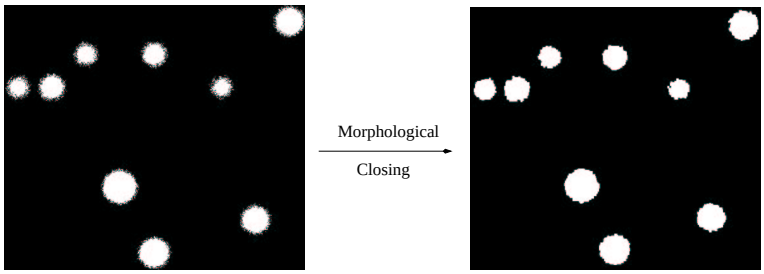

Figure 10: Effect of a morphological closing

selected pixels are set to 0 and pixels of each separate object are set to a label value. In this example, the labels go from 1 to 9.

Such an integer image is a convenient tool to apply some computations on one and only one specific object at a time.

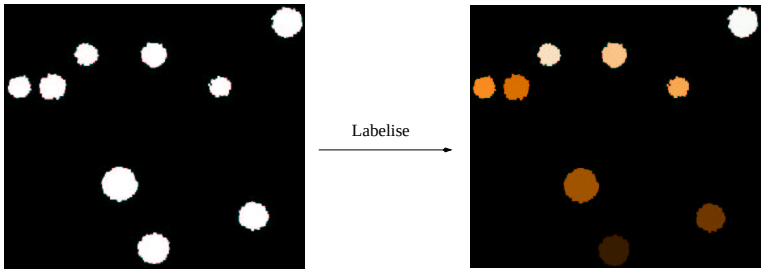

Figure 11: Labelisation of a binary image

Binary images are not supposed to contain any bad pixel map. If there is one attached to the binary image, it would be simply ignored.

The `cpl_image_binary-test.c` file in the CPL `tests` directory contains examples of `cpl_image_binary` function usage.

5.2.2 Tables

Tables are generally defined as rectangular arrangements of cells, where cells belonging to the same column contain data of the same type, while cells from the same row are related by some unifying characteristics. The *cpl_table* component is based strictly on this definition.

A *cpl_table* is made of columns, and a column consists of an array of elements of a given type. Currently, three numerical types are supported, `CPL_TYPE_INT`, `CPL_TYPE_FLOAT`, and `CPL_TYPE_DOUBLE`, plus a type indicating columns made of character strings, `CPL_TYPE_STRING`. However, because of the way it is designed, the *cpl_table* could be easily extended in future to support table columns of any conceivable data type defined within the CPL (for instance, tables with columns whose cells would contain *cpl_vectors* or *cpl_images* may be created).

A table column should only be accessed through the *cpl_table* interface, by specifying its name. The ordering of the columns within a table is undefined; a *cpl_table* is not a n-tuple of columns, but just a set of columns. The N elements of a column are counted from 0 to $N - 1$, with element 0 on top. The set of all the table columns' elements with the same index constitutes a table row, and table rows are counted according to the same convention. It is possible to flag each *cpl_table* row as 'selected' or 'unselected', and each column element as 'valid' or 'invalid' (*null* flagged). Selecting table rows is mainly a way to extract just those table parts fulfilling any given condition, while invalidating column elements is a way to exclude such elements from any computation.

The *cpl_table* component ensures optimal performance and memory handling for most purposes. However, a pointer to the primitive data types contained in a specific column or cell may be obtained, whenever the developer finds that some table system performance drawback needs to be overcome.

A *cpl_table* may be created by means of its specific constructors, and used for storage and handling of information that was generated within a program. The code in this case may look like this (error checking is omitted for clarity):

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    int        number_of_rows;
    ...
    ...
    table = cpl_table_new(number_of_rows);

    cpl_table_new_column_string(table, "Player");
    cpl_table_new_column_int(table, "Games won");
    cpl_table_new_column_int(table, "Games lost");
    cpl_table_new_column_float(table, "Success rate");
    ...
    cpl_table_delete(table);
    ...
    return 0;
```

```
}
```

Alternatively, a *cpl_table* may be simply loaded from a FITS file table extension, as in the following example:

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    int        number_of_rows;
    ...
    ...

    /*
     *  Loading a table from extension 2 of a FITS file.
     *  The last argument indicates that invalid table elements should
     *  be flagged.
     */

    table = cpl_table_load("Championship_2003.fits", 2, 1);
    number_of_rows = cpl_table_get_nrow(table);
    ...

    /*
     * Write the processed table to disk in FITS format (using a default
     * FITS header), clean memory, then exit.
     */

    cpl_table_save(table, NULL, NULL, "Revised_Championship_2003.fits", 0);
    cpl_table_delete(table);
    ...
    return 0;
}
```

The following operations can be performed through the *cpl_table* methods' interface:

- defining and allocating new columns,
- creating new columns pointing to existing arrays of data,
- reading and writing table cells,
- shifting positions of column values,
- supporting invalid table cells,
- computing statistical quantities, performing arithmetic with scalar columns, etc., excluding invalid cells from the computations,

- exporting column data, assigning a code of choice to invalid numerical cells,
- column duplication, casting, moving from one table to another,
- resizing tables,
- merging tables,
- duplicating tables,
- creating new tables modelled on existing tables,
- sorting table rows,
- selecting and extracting subtables from existing tables, and,
- loading and saving tables as FITS files.

The methods to support these and other operations are all described in detail in the *CPL Reference Manual* [4] but, in the following, some of the functionalities are explained with the help of a number of simple examples.

1. **Support of invalid table cells**

   Table cells may be flagged as invalid. This is, in general, a way to exclude some of the values from a given operation, for instance the computation of a mean, or of an arithmetic operation, as in the following example (error checking is omitted for clarity):

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    int        i;
    int        nrows = 10;
    double     mean;
    ...


    /*
     * Create a table with a predefined length of 10 rows, and create
     * an integer column named "Numbers" with the numbers from 1 to 10:
     */

    table = cpl_table_new(nrows);

    cpl_table_new_column_int(table, "Numbers");
    for (i = 0; i < nrows; i++)
        cpl_table_set_int(table, "Numbers", i, i + 1);
```

```
    /* Flag the "Numbers" column's first and third cells as invalid */

    cpl_table_set_null(table, "Numbers", 0);
    cpl_table_set_null(table, "Numbers", 2);

    /*
     * Compute the mean value: the values flagged as invalid are
     * automatically excluded from the computation:
     */

    mean = cpl_table_column_mean(table, "Numbers");

    /*
     * Now restore the original values: the corresponding table
     * column elements are automatically considered again as valid.
     * A different mean value is now computed.
     */

    cpl_table_set_int(table, "Numbers", 0, 1);
    cpl_table_set_int(table, "Numbers", 2, 3);

    mean = cpl_table_column_mean(table, "Numbers");

    ...
    cpl_table_delete(table);
    ...
    return 0;
}
```

It should be noted that when a table column value is flagged as *invalid*, it is lost: there is no function to unset a *null* flag. The only way to unset a *null* flag is to write a valid value to the corresponding table cell. It is important to be aware of this every time the data array of a table column is exported to another process (*e.g.*, a fitting routine), as in the following code section:

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    float     *data;
    int        size;
    ...

    /*
     * It is assumed that the float column "Data" contains some
     * invalid values. The data array of the table column is extracted
     * and passed to an external fitting routine, but this is a
     * mistake: in fact the array elements corresponding to an
```

```
     * invalid element contain garbage.
     */

    data = cpl_table_get_data_float(table, "Data");
    size = cpl_table_get_nrow(table);

    <result of the fit> = fit(data, size);

    /*
     * In case the external fitting routine would support a special
     * "code" to identify invalid values that would be excluded from
     * the fit - for instance, 0.0 - such code may be written to the
     * internal data buffer before exporting:
     */

    cpl_table_column_code_null_float(table, "Data", 0.0);

    /*
     * In this way the invalid values would still remain flagged as
     * invalid, but the exported data would not contain any garbage
     * and the fitting routine would work properly:
     */

    data = cpl_table_get_data_float(table, "Data");
    size = cpl_table_get_nrow(table);

    <result of the fit> = fit(data, size);

    /*
     * It is likely that a more common solution would be to physically
     * remove any invalid value from a table before exporting the
     * internal data buffer to the foreign routine. Here the table
     * would be modified, and its size would be smaller than before:
     * the function cpl_table_clean() removes from a table any row
     * containing at least one invalid value.
     */

    cpl_table_clean(table);
    data = cpl_table_get_data_float(table, "Data");
    size = cpl_table_get_nrow(table);

    <result of the fit> = fit(data, size);
    ...
    cpl_table_delete(table);
    ...

    return 0;
}
```

The most obvious example of exporting a column’s internal data buffer to an external process is when a ta-

ble is converted to FITS format and written to disk. This is done by the function `cpl_table_save()`, that converts any invalid column value to the FITS convention for *null* values. Invalid values in numerical columns of type `CPL_TYPE_FLOAT` and `CPL_TYPE_DOUBLE` are replaced by their own `NaN` bit pattern, while invalid character strings in `CPL_TYPE_STRING` columns are replaced by sequences of blanks. The only exception is represented by invalid values in columns of type `CPL_TYPE_INT`, that are the only ones that need a specific code to be explicitly assigned to them. This can be realised by calling the function `cpl_table_column_code_null_int()` for each table column of type `int` containing invalid values, just before saving the table to FITS. The numerical values identifying invalid integer column elements are written to the FITS keywords `TNULL`*n* (where n is the column sequence number).

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    int        nrows = 10;
    ...


    /*
     * Create a table with a predefined length of 10 rows, create
     * an integer column named "Numbers", and fill it with the value 3:
     */

    table = cpl_table_new(nrows);

    cpl_table_new_column_int(table, "Numbers");
    cpl_table_fill_column_int(table, "Numbers", 0, nrows, 3);

    /* Flag the "Numbers" column's first and third cells as invalid */

    cpl_table_set_null(table, "Numbers", 0);
    cpl_table_set_null(table, "Numbers", 2);

    /*
     * Save to a FITS file, but give first the code 999 for the NULL
     * values. The output FITS file header will contain the TNULL
     * keyword (corresponding to this column) set to 999.
     */

    cpl_table_column_code_null_int(table, "Numbers", 999);
    cpl_table_save(table, NULL, NULL, "output_table.fits", 0);
    cpl_table_delete(table);
    ...

    return 0;
}
```

Beware that if valid column elements have the value identical to the chosen *null*-code, they will mistakenly be considered invalid within the FITS convention.

2. **Shifting position of column values**

   It may be useful in some cases to shift the positions of all the values of a given table column by a specified amount. This is done with the table function `cpl_table_shift_column()`. The most obvious application of this functionality is in the computation of the finite differences of a sequence of numbers, the discrete analogue of the differential operation.

   In the following example the finite forward difference of the values in the `float` table column `"Values"` is written to the new `float` table column `"Forward differences"` (error checking is omitted for clarity):

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    char       input[] = "input_table.fits";
    char       output[] = "output_table.fits";
    ...


    /*
     * Load the table data from a given FITS file. We assume here
     * that the table contains a float column named "Values".
     */

    table = cpl_table_load(input, 1, 1);

    /*
     * A simple procedure: duplicate the input column, move the values
     * of the duplicated column upward by one position, and finally
     * subtract the original column values from the shifted ones,
     * writing the result to the duplicated column itself.
     */

    cpl_table_duplicate_column(table, "Forward differences", table, "Values");
    cpl_table_shift_column(table, "Forward differences", -1);
    cpl_table_subtract_columns(table, "Forward differences", "Values");

    /*
     * Write the new table to disk in FITS format (using a default FITS
     * header), clean memory, then exit.
     */
```

```
    cpl_table_save(table, NULL, NULL, output, 0);
    cpl_table_delete(table);

    return 0;
}
```

In the example the last element of the `"Forward differences"` column turns out to be flagged as *invalid*: the upward shift leaves the corresponding table cell empty, so that it was automatically excluded by the subtraction operation.

3. **Selecting and extracting subtables from existing tables**

   A set of functions of the *cpl_table* component is used to select a number of rows from an existing table, before copying them to a new table. The selection functions are used to apply simple selection criteria, that can be logically combined to define more complex criteria. With the only exception of the function `cpl_table_reverse_selection()`, all the selection functions names include the words `_and_` or `_or_`, to indicate how a given selection criterion should be combined with the existing row selection of a given table. The `_and_` tag indicates that between the existing selection and the new selection criterion an *intersection* is made, while the `_or_` tag indicates that between the existing selection and the new selection criterion a *union* is made. The initial state of any table is that all of its rows are selected, and therefore the first selection applied to a table would always be an `_and_` selection, as shown in the following example:

```
...
#include <cpl_table.h>
...
int main()
{
    ...
    cpl_table *table;
    cpl_table *subtable;
    char       input[] = "input_table.fits";
    char       output[] = "output_table.fits";
    int        selected;
    ...


    /*
     * Load the table data from a given FITS file. We assume here
     * that the table contains a float column named "Day", a string
     * column named "Month", and an integer column named "Year".
     * This table begins with all rows selected, but in this
     * example we ensure this explicitly:
     */

    table = cpl_table_load(input, 1, 1);
    cpl_table_select_all(table);          /* Not really necessary... */

    /*
```

```
     * Here we select all rows containing the year 1958 and the year
     * 2003; from those we select those having a month beginning with
     * the letter "A" or "a", and a day between 5.5 (included) and 12.3
     * (excluded). Finally, we add to all these any row containing
     * the month "May" (no matter what year or day). Each function
     * call returns the total number of selected rows, that in this
     * example is always discarded, with the exception of the last
     * call.
     */

    cpl_table_and_select_int(table, "Year", EQUAL_TO, 1958);
    cpl_table_or_select_int(table, "Year", EQUAL_TO, 2003);
    cpl_table_and_select_string(table, "Month", EQUAL_TO, "^[Aa].*");
    cpl_table_and_select_float(table, "Day", NOT_LESS_THAN, 5.5);
    cpl_table_and_select_float(table, "Day", LESS_THAN, 12.3);
    selected = cpl_table_or_select_string(table, "Month", EQUAL_TO, "May");

    /*
     * If some rows survived, a new table is created from the selected
     * rows and it is saved to a FITS file:
     */


    if (selected != 0) {
        subtable = cpl_table_extract_selected(table);
        cpl_table_save(subtable, NULL, NULL, output, 0);
        cpl_table_delete(subtable);
        cpl_table_delete(table);
        return 0;
    }

    cpl_table_delete(table);
    return 1;

}
```

Note that in matching strings the reference value is interpreted as a regular expression. All the selection functions involving comparisons with a constant require that the constant has the same type of the referred column. For this reason there is a function for each available column type. The functions `cpl_table_and_select()` and `cpl_table_or_select()`, without any type suffix, are used in the comparison of the values from two numerical columns.

5.2.3 Vectors

In the *Common Pipeline Library*, the vector component is named *cpl_vector*. It is a simple structure with an array of *double* values and a size. This basic object can be used to build more complicated types, such as a complex array (combination of a vector for the real values and a vector for the imaginary values) or a 1-dimension function (see 5.2.4).

To create or delete a *cpl_vector* object, you must use the dedicated functions *cpl_vector_new()* and *cpl_vector_delete()*.

Here is an example that shows how a *cpl_vector* can be used to load a values list from a text file, to subtract the mean and write the result into another text file:

```
int main()
{
    cpl_vector *vect;
    double      mean;
    FILE       *out;

    /*
     * Read values from an ASCII file and store it in a cpl_vector.
     * myfile.txt contains a list of the vector values (one per line)
     */

    vect = cpl_vector_read("myfile.txt");

    /* Compute the mean of the vector */
    mean = cpl_vector_mean_reject(vect, 1, 1);

    /* Subtract the mean */
    cpl_vector_const_op(vect, mean*(-1.0), '-');

    /* Write out the result to a file */
    out = fopen("output_file.txt", "w");
    cpl_vector_dump(vect, out);

    /* Delete and close */
    cpl_vector_delete(vect);
    fclose(out);

    /* Return */
    return 0;
}
```

Some of the functionalities provided by this component are :

- vector constructor and destructor,
- routines to read/write a vector from/to a file,
- sorting functionality,
- basic arithmetic operations between vectors or between a vector and a constant,
- statistics computed on a vector (find the minimum, the maximum, calculate the mean, ...),
- derive the low frequency signal from a vector, and,
- vectors comparison methods.

The functionalities implemented at the moment are basic. The aim is not to try to forsee every conceivable function that could be needed. If new requirements come, then the dedicated functions will be designed accordingly. This approach keeps the *Common Pipeline Library* as small as possible, but not excluding the possibility of later extension.

5.2.4 1d Functions

In addition to the vector component, a 1-dimension function component, *cpl_1dfunction*, has been defined. This object is simply composed of two *cpl_vector* objects; one for the X coordinate, the other for Y.

Again, there are dedicated functions to create or destroy this object (in this case, *cpl_1dfunction_new()* and *cpl_1dfunction_delete()*).

The functionality provided by the 1d Function methods includes:

- a constructor and a destructor,
- read/write functionalities, and,
- local maximum, centroiding, interpolation or cross-correlation functions.

5.2.5 Matrices

Matrices are generally defined as a set of numbers arranged in a rectangular grid of rows and columns. The *cpl_matrix* component only supports sets of numbers in double precision.

The *cpl_matrix* is an opaque object; access and manipulation of matrix data is done through an interface of methods and accessors designed for that purpose. Such methods are intended to support basic matrix handling, ensuring optimal performance and memory usage. Besides, a pointer to the data buffer of matrix elements is available whenever the developer finds that a particular algorithm is missing from the library, or specific performance requirements need to be fulfilled. The internal data buffer of a *cpl_matrix* is a simple array of `double` values, where the first value refers to the upper left position of the matrix, and the last value to the lower right position. The values are listed row by row, with each row running from left to right and starting with the top row. The elements of a *cpl_matrix* are indexed starting from 0, *i.e.*, the first matrix element at the upper left position has index $0, 0$.

A *cpl_matrix* may be created with one of its specific constructors, and used for storage and handling of information that was generated within a program. The code may look like this (error checking is omitted for clarity):

```
...
#include <cpl_matrix.h>
...
int main()
{
    ...
    cpl_matrix *matrix;
```

```
    double     *data_buffer;
    int         number_of_rows    = 20;
    int         number_of_columns = 4;
    double      value;
    ...

    ...
    matrix = cpl_matrix_new(number_of_rows, number_of_columns);
    ...

    /* Copy the value of a matrix elements to another location */

    value = cpl_matrix_get(matrix, 0, 3);
    cpl_matrix_set(matrix, 4, 1, value);
    ...

    /*
     *  Direct access to the matrix data buffer: the buffer
     *  always contains double precision values...
     */

    data_buffer = cpl_matrix_get_data(matrix);
    ...
    cpl_matrix_delete(matrix);
    ...
    return 0;
}
```

Currently *cpl_matrix* supports the following operations with matrices:

- creating different types of matrices, duplicating matrices, etc.,
- reading and writing matrix elements,
- transposing, shifting, removing row/column intervals, and performing any other elementary row/column operations,
- extracting submatrices, expanding existing matrices, merging of matrices,
- performing arithmetic, computing scalar products, determinants, etc.,
- computing statistical quantities,
- sorting of matrix rows or columns, gaussian elimination, etc.,
- solving systems of linear equations, and,
- inversion.

The methods to support these and other operations are all described in detail in the *CPL Reference Manual* [4], but in the following some of the functionalities are explained with the help of one single example, adapted from

a higher-level function of the *cpl_matrix* component, `cpl_matrix_leastsq()`. This function is used to solve redundant linear systems, *i.e.*, linear systems with too many equations or too many unknowns.

The theory: given the matrix of the linear system coefficients **C**, and the non-homogeneous term **H**, the system

$$\mathbf{C} \cdot \mathbf{X} = \mathbf{H}$$

is defined, where **X** is the column matrix of the unknowns. The pseudo-inverse solution of this system is given by

$$\mathbf{X} = \mathbf{H} \cdot \mathbf{C}^T \cdot inv(\mathbf{C} \cdot \mathbf{C}^T))$$

where $\mathbf{C}^T$ represents the transposed matrix of **C**, and inv the matrix inversion operation. In the following code, a system of 100 equations in 10 unknowns is solved:

```
...
#include <cpl_matrix.h>
...
int main()
{
    ...
    cpl_matrix *coeff;
    cpl_matrix *t_coeff;
    cpl_matrix *nonhomo;
    cpl_matrix *solution;
    cpl_matrix *m1;
    cpl_matrix *m2;
    cpl_matrix *m3;
    int         equations = 100;
    int         unknowns = 10;
    int         i, j;
    ...


    /* Creating the coefficient and the non-homogeneous term matrices */

    coeff = cpl_matrix_new(equations, unknowns);
    nonhomo = cpl_matrix_new(equations, 1);

    /*
     * The matrices are filled in some way with the appropriate data,
     * for instance using the function cpl_matrix_set():
     */


    ...
    cpl_matrix_set(coeff, i, j, value);
    ...
    cpl_matrix_set(nonhomo, i, 1, value);
    ...

    /* Now that the matrices are available we can apply the theory */
```

```
    t_coeff = cpl_matrix_transpose(coeff);
    m1 = cpl_matrix_product(coeff, t_coeff);
    m2 = cpl_matrix_inverse(m1);
    if (m2 == NULL)                         /*  Singular matrix  */
        return 1;
    m3 = cpl_matrix_product(nonhomo, t_coeff);
    solution = cpl_matrix_product(m3, m2);

    /* Some cleaning... */

    cpl_matrix_delete(coeff);
    cpl_matrix_delete(nonhomo);
    cpl_matrix_delete(t_coeff);
    cpl_matrix_delete(m1);
    cpl_matrix_delete(m2);
    cpl_matrix_delete(m3);

    /*
     * Here the solution is available and can be used. Finally, also
     * the solution matrix is deleted and the program closed:
     */

    ...
    cpl_matrix_delete(solution);
    ...
    return 0;
}
```

5.2.6 Messaging and logging

A simple component for displaying informative text to terminal and for maintaining logfiles is available in the CPL. The following operations are supported:

- controlling whether or not messages are written to the terminal and/or to a logfile,
- optionally adding informative tags to messages,
- setting width for message line wrapping,
- controlling the message indentation level, and,
- filtering messages according to their severity level.

Messages may be printed using any of the following functions:

- `cpl_msg_debug()`,
- `cpl_msg_info()`,

- `cpl_msg_warning()`, and,
- `cpl_msg_error()`.

Choosing from these functions means assigning a level of severity to a given message. The messaging system can then be set to display just messages having sufficient severity, choosing a verbosity level from the following list:

- `CPL_MSG_DEBUG`,
- `CPL_MSG_INFO`,
- `CPL_MSG_WARNING`,
- `CPL_MSG_ERROR`, and,
- `CPL_MSG_OFF`.

The highest verbosity level of the messaging system is `CPL_MSG_DEBUG`. That would ensure that *all* the messages are printed. The verbosity would progressively decrease through the levels `CPL_MSG_INFO`, `CPL_MSG_WARNING`, and `CPL_MSG_ERROR`, where only messages served by the `cpl_msg_error()` function would be printed. The lowest verbosity level, `CPL_MSG_OFF`, would inhibit the printing of any message to the terminal.

To activate and deactivate the messaging system, the functions `cpl_msg_start()` and `cpl_msg_stop()` should be used. These functions will typically be called at the beginning and at the end of a program, and an attempt to use an uninitialised messaging system would generate an assertion failure. To output the messages to a logfile, a call to `cpl_msg_log_on()` is also required, while output to terminal is automatically enabled at a verbosity level `CPL_MSG_INFO`; the function `cpl_msg_terminal_on()` may be used just to modify this default verbosity.

Three different tags may be attached to any message: *time*, *domain*, and *component*. The *time* tag is the time of the printing of the message, and can optionally be turned on or off with the functions `cpl_msg_time_tag_on()` and `_off()`. The *domain* tag is an identifier of the main program (typically, a pipeline recipe), and can be optionally turned on or off with the functions `cpl_msg_domain_tag_on()` and `_off()`. Finally, the *component* tag is used to identify a component of the program (typically, a function), and can be optionally turned on or off with the functions `cpl_msg_component_tag_on()` and `_off()`. However, the *component* tag is always shown when the verbosity level is set to `CPL_MSG_DEBUG`.

As a default, none of the above tags are attached to messages sent to the terminal, but all the tags are always shown in messages sent to the logfile. A further tag, the *severity* tag, can never be turned off. This tag depends on the function used to print any given message. The tags are prepended to all messages, and are not affected by the message indentation controlled by the functions `cpl_msg_indent()`, `cpl_msg_indent_more()`, `cpl_msg_indent_less()`, and `cpl_msg_set_indent_step()`.

The messaging component takes care of adapting long lines of text to the actual terminal width or to a specific maximum value, adding a new line character at the end of any message. Inserting new line characters would enforce breaking a line of text even before the current row is filled.

In the following, an illustration of writing messages to terminal and to a logfile is given.

```
...
#include <cpl_messaging.h>
...
int main()
{
    ...
    char domain[] = "Example";
    char component[] = "messaging";
    ...

    /* Initialising the messaging system. */

    cpl_msg_start();
    cpl_msg_terminal_on(CPL_MSG_WARNING);
    cpl_msg_log_on(CPL_MSG_DEBUG);
    cpl_msg_time_tag_on();
    cpl_msg_component_tag_on();
    cpl_msg_set_domain_tag(domain);
    cpl_msg_domain_tag_on();

    /* Sending messages both to terminal and to logfile. */

    cpl_msg_debug(component, "Log is written to %s", cpl_msg_log_file());
    cpl_msg_info(component, "This is message number %d of %d", 2, 4);
    cpl_msg_warning(component, "This is a %s message", "warning");
    cpl_msg_error(component, "This is the final error message");
    ...
    cpl_msg_stop();

    return 0;
}
```

A complete description of the functions available in the messaging component is given in the *CPL Reference Manual* [4].

5.2.7 Error handling

This component provides a means to detect the occurrence of errors, and the supporting of functions to diagnose or obtain additional information when such errors occur.

A *cpl_error_code* is set by any CPL function, similarly to what is done with the *errno* variable of the standard C library. The following guidelines are respected:

- if no error occurs in a CPL function, the *cpl_error_code* will remain unchanged, and,
- an error condition in a CPL function will cause the previous *cpl_error_code* to be overwritten by the new one.

A *cpl_error_code* equal to the enumeration constant `CPL_ERROR_NONE` would indicate no error condition. Note, however, that the *cpl_error_code* is only set when an error occurs, and it is not reset by successful function calls. For this reason it may be appropriate in some cases to reset the *cpl_error_code* using the function `cpl_error_reset()` (as shown in the examples below). The *cpl_error_code* set by a CPL function can be obtained by calling the function `cpl_error_get_code()`. Functions of type *cpl_error_code* would not only return this code directly, but would also return `CPL_ERROR_NONE` in case of success. Other CPL functions return zero on success, or a non-zero value to indicate a change of the *cpl_error_code*, while CPL functions returning a pointer would flag an error by returning a `NULL`.

To each *cpl_error_code* is associated a standard, human-readable error message, that can be obtained by calling the function `cpl_error_get_message()`. Conventionally no CPL function will ever display any error message, leaving to the caller the decision of how to handle a given error condition. A call to the function `cpl_error_get_function()` would return the name of the function where the error occurred, and the functions `cpl_error_get_file()` and `cpl_error_get_line()` would also return the name of the source file containing the function code, and the line number where the error occurred. The function `cpl_error_get_where()` would gather all these items together, in a colon-separated string.

The currently available error codes are:

- CPL_ERROR_NONE

  No error condition.

- CPL_ERROR_DUPLICATING_STREAM

  Cannot duplicate output stream.

- CPL_ERROR_ASSIGNING_STREAM

  Cannot associate a stream with a file descriptor.

- CPL_ERROR_FILE_IO

  File access failed due to insufficient read or write permission.

- CPL_ERROR_BAD_FILE_FORMAT

  An input file did not have the expected format, for instance FITS.

- CPL_ERROR_FILE_ALREADY_OPEN

  An attempt was made to open an already open file that should be opened just once, as is the case with the CPL logfile created by the related CPL messaging functions.

- CPL_ERROR_FILE_NOT_CREATED

  An attempt to create a file on disk failed.

- CPL_ERROR_FILE_NOT_FOUND

  A file was not found on disk.

- CPL_ERROR_DATA_NOT_FOUND

  A component of a valid object was not found (could be a column within a table).

- CPL_ERROR_ACCESS_OUT_OF_RANGE

  An object was accessed beyond its boundaries.

- CPL_ERROR_NULL_INPUT

  A *NULL*-pointer was received where a valid pointer was expected.

- CPL_ERROR_INCOMPATIBLE_INPUT

  Two input arguments were incompatible. This may occur on operations involving multiple datasets (performing arithmetic, merging, etc.).

- CPL_ERROR_ILLEGAL_INPUT

  An input argument was illegal, *e.g.*, trying to create a matrix having negative sizes.

- CPL_ERROR_ILLEGAL_OUTPUT

  A CPL function generated a return value that would generate an illegal object (as a matrix consisting of just one element, or a zero length table).

- CPL_ERROR_UNSUPPORTED_MODE

  A requested functionality is not actually supported.

- CPL_ERROR_SINGULAR_MATRIX

  Matrix inversion was attempted (implicitly or explicitly) with a singular matrix.

- CPL_ERROR_DIVISION_BY_ZERO

  An attempt was made to divide by zero, or to perform a division that would lead to numerical overflow.

- CPL_ERROR_TYPE_MISMATCH

  An argument had the wrong type for the attempted operation, as in the case of trying to write a numerical value in a string table column.

- CPL_ERROR_INVALID_TYPE

  Data type is not supported.

Here is an example that uses the CPL error functions for detecting and handling possible error conditions coming from different kinds of functions:

```
...
#include <cpl_error.h>
#include <cpl_messaging.h>
...
int main()
{
    ...
    cpl_matrix     *matrix = cpl_matrix_new(10, 10);
    cpl_matrix     *inverse;
    int             start, count;
```

```
cpl_error_code  status;
char           *message;
char           *component;
char           *where;
...

/*
 * Handle failure of a function returning a valid pointer
 * on success, or a NULL in case of failure. The function
 * and the standard message associated with the error code
 * are retrieved and used.
 */

inverse = cpl_matrix_inverse(matrix);

if (inverse == NULL) {
    cpl_msg_error(cpl_error_get_function(), cpl_error_get_message());
    return cpl_error_get_code();
}
...

/*
 * Handle failure of a function returning 0 (CPL_ERROR_NONE)
 * on success and a non-zero error code in case of failure.
 * In case of an error a switch may be done on the error code.
 * The error-message/warning will display a combination of the
 * function name, the file of the source code and its line number.
 */

status = cpl_matrix_delete_rows(matrix, start, count);

if (status != CPL_ERROR_NONE) {

    message = cpl_error_get_message();
    where = cpl_error_get_where();
    component = cpl_error_get_function();

    switch (status) {
    case CPL_ERROR_ACCESS_OUT_OF_RANGE:
        cpl_msg_warning(component, message);
        cpl_error_reset();
        break;
    default:
        cpl_msg_error(where, message);
        return status;
    }

}
...

/*
```

```
        * Handle failure of a function whose return value cannot indicate
        * the error status. Here the error-messaging uses separate parts
        * for the function name, the file of the source code and its line
        * number.
        */

       mean = cpl_matrix_mean(matrix);
       status = cpl_error_get_code();

       if (status != CPL_ERROR_NONE) {
           component = cpl_error_get_function();
           cpl_msg_error(component, cpl_error_get_message());
           cpl_msg_error(component, "at line %u", cpl_error_get_line()) ;
           cpl_msg_error(component, "of file %s", cpl_error_get_file()) ;
           return status;
       }
       ...
       return 0;

   }
```

The few functions to support error handling are all described in detail in the CPL Reference Manual [4].

5.3 Properties

A *cpl_property* is a name/value pair used for storing meta-data. Although this facility is made available to the programmer for implementing his or her own data structures, it is expected that the “property list” facility would be used in most applications requiring this sort of functionality (see Section 5.3.1). Note the difference between a *cpl_property* (an atomic variable storage mechanism) and a *cpl_plist* (which organises and stores complete sets of associated variables).

The *cpl_property* supports several different primitive datatypes for the stored value. In particular, all the types foreseen by the FITS standard for header keywords are provided. A single complex datatype, namely that of strings, is also available.

As the values of properties are stored in binary form, a property can be used as lossless storage for such named parameters within the application. This eliminates the concern of loss of information due to conversion to, for example, text strings, etc..

In addition to the name and value, it is possible to associate a descriptive comment with the property. This comment could be used to store explanatory text, information about units or whatever is required. Note that there is no explicit field for the units within the property itself.

5.3.1 Property lists

The property list facility provided by the CPL offers a way to store meta-data as a sequence of name/value pairs. Although the internals of the *cpl_plist* make use of the *cpl_property* type (see Section 5.3), the property

list interface completely hides this detail, and allows the user to manipulate his or her data through a single interface. Thus, unlike parameter lists, it is not possible (or even necessary) to extract/insert properties from the property list.

The *cpl_plist* was designed for supporting the FITS header information. Indeed, it is possible, using a single function, to load a header file into a property list, given the filename and the number of the extension.

To obtain a value from a property list, the list is queried by looking for the value's name as shown below. New values can be added to a property list and entries can be erased. But, *properties* which belong to a property list exist only as part of this list, *i.e.* , a property cannot be extracted from a property list.

```
#include <cpl_plist.h>

...

int main()
{

  ...

  int i, status;
  float f;
  char *s;


  cpl_plist *list;

  ...

  list = cpl_plist_new();

  ...

  cpl_plist_append_int(list, "MyInt", 42);
  cpl_plist_append_float(list, "MyFloat", 1.e-6);
  cpl_plist_append_string(list, "MyString", "text");

  ...

  i = cpl_plist_get_int(list, "MyInt");
  f = cpl_plist_get_float(list, "MyFloat");
  s = cpl_plist_get_string(list, "MyString");

  ...

  cpl_plist_delete(list)

  ...

  return 0;
```

```
}
```

Within the CPL, property lists are used to store the headers of FITS files. The translation from and to a FITS header is done on the fly.

5.4 Standard data reduction algorithms in *libcplbase*

The CPL *libcplbase* library will provide standard astronomical data reduction algorithms in the future. This library is not yet available.

5.5 The CPL high-level interfaces in *libcplui*

5.5.1 Frames

A *cpl_frame* is a way of associating attributes to files. It is used as a communication method between a data reduction organiser and a data reduction task. Because multiple data files are often required in the processing of a single observation (dark, flat, bias, target, etc.), it is often necessary to associate these different files for any data reduction task. The frame component of the CPL makes this possible.

Among the data set attributes are the filename to which the frame is associated, its type, the group to which it belongs and, if the frame describes a processing product, possibly a processing level.

The *cpl_frame* component provides the functions to set and query frame attributes, as shown in the example below:

```
#include <cpl_frame.h>

...

cpl_frame *add(cpl_image *image1, cpl_image *image2)
{

    cpl_frame *product_frame;


    cpl_image_add_local(image1, image2);


    product_frame = cpl_frame_new();

    cpl_frame_set_filename(product_frame, "image12.fits");
    cpl_frame_set_tag(product_frame, "ADDED_IMAGE");
    cpl_frame_set_type(product_frame, CPL_FRAME_TYPE_IMAGE);
    cpl_frame_set_group(product_frame, CPL_FRAME_GROUP_PRODUCT);
    cpl_frame_set_level(product_frame, CPL_FRAME_LEVEL_FINAL);
```

```
    return product_frame;

}
```

5.5.2 Frameset

A frameset is just a container for frames. Frames can be added to a frameset and can be looked up by a tag or by sequentially traversing the container. The frameset is part of the CPL recipe plugin interface (see Section 3.5). In this context, it is used to pass input files to a data reduction task and obtain the products from it after it has been completed.

```
#include <cpl_frameset.h>

...

cpl_frameset *subtract_bias(cpl_image *image, cpl_frameset *set)
{

    ...

    cpl_frame *bias_frame,
    cpl_frame *result_frame;

    cpl_image *bias = cpl_image_new_double(2048, 4096, NULL, NULL);

    ...

    bias_frame = cpl_frameset_find(set, "BIAS");
    bias = cpl_image_copy_from_fits(bias,
                                    cpl_frame_get_filename(bias_frame),
                                    0, 0);
    ...

    result_frame = cpl_frame_new();

    ...

    cpl_frameset_insert(set, result_frame);

    ...

    return set;

}
```

5.5.3 Parameters

A parameter is a datatype with an associated name, description and value-checking. Parameters are designed to handle monitor/control data and they provide a standard way to pass for instance command line information to different components of an application.

The implementation supports three classes of parameters: a plain value, a value within a given range, or an value as part of an enumeration. When a parameter is created it is created for a particular value type. In the latter two cases, validation is performed whenever the value is set.

The type of a parameter's current and default value may be: boolean, integer, double or string.

In addition to the name, parameters provide an associated context. Parameter names must be unique — they define the identity of a given parameter. The context is used to associate parameters together. A context, for example, may be the name of the part of the application, from where the parameter value originated.

Parameters were designed to be used by the PDRM interface, as a method of passing command data between a host application and a recipe.

Parameters vary from properties, in that they have these associated data constraints and additional descriptive parameters. While properties are primitive units of data storage without any overhead, parameters offer self-description and data integrity checking which are essential for dealing with interfaces within the application.

Parameters may be grouped using the "parameter list" component. A parameter list, *cpl_plist*, is simply a mechanism for grouping lists of parameters. It provides a convenient way for passing large numbers of parameters to a function. For instance, it is used in the plugin interface to pass the parameters a recipe accepts from the plugin to the calling application and vice versa.

Unlike the relationship between properties and property lists, it is possible to extract/insert parameters within parameter lists.

For a complete documentation of the parameter component please refer to the CPL Reference Manual [4].

```
#include <cpl_parameter.h>
#include <cpl_parlist.h>

...

cpl_parlist *make_parameter_list(int i, double d, const char *s)
{

    cpl_parlist *plist = cpl_parlist_new();
    cpl_parameter *p;


    p = cpl_parameter_value_new("config.integer_value",
                                CPL_TYPE_INT,
                                "An integer value",
                                "config",
                                0);

    cpl_parameter_set_int(p, i);
```

```
    cpl_parlist_append(plist, p);

    p = cpl_parameter_range_new("config.double_range",
                                CPL_TYPE_DOUBLE,
                                "An range of doubles",
                                "config",
                                0.5, 0., 1.);

    cpl_parameter_set_double(p, d);
    cpl_parlist_append(plist, p);

    p = cpl_parameter_enum_new("config.string_enum",
                                CPL_TYPE_STRING,
                                "An enumeration of strings",
                                "config",
                                "one", 3, "one", "two", "three");

    cpl_parameter_set_string(p, s);
    cpl_parlist_append(plist, p);

    return plist;

}
```

A Memory model description

In principle, the same memory handling functions should be used everywhere in a piece of software. The actual rule is that all memory allocated with a given model has to be deallocated using the same one.

If the CPL is used in your code, all the memory returned by CPL functions must be de-allocated with an `xmemory` deallocation function. For all CPL objects allocated/returned by CPL functions, a destructor is provided. Using a CPL function to de/allocate memory ensures that the `xmemory` model is used.

Example:

```
cpl_image *img;

/* Image allocated by a CPL function (with the xmemory model) */
img = cpl_image_new_float(1024, 512);
...

/* Image deallocated by a CPL function (with the xmemory model) */
cpl_image_delete(img);
```

In the cases where a CPL function returns a newly allocated array, you may be tempted to use a simple `free()` call to deallocate the array, which would be a mistake; you must use the `xmemory` model freeing function (see Problem 1).

Problem 1:

```
cpl_image *obj;
double    *values;

/* Values allocated by a CPL function (with the xmemory model) */
values = cpl_image_get_values(obj);
...

/* Values deallocated (without the xmemory model) */
free(values);              /* ERROR! */
```

To avoid any problems, the easiest solution is to make sure that the `xmemory` functions are used everywhere in your code.

To do that, there are two solutions:

Solution 1: You just have to include `xmemory.h` in your source files to make sure that your calls to allocation functions will be, in reality, calls to the `xmemory` functions. In this case, the process is completely invisible, and you can continue to use `malloc()`, etc., without thinking any more about memory allocation problems.

Problem 1 solved with Solution 1:

```
#include "xmemory.h"
...
```

```
cpl_image *obj;
double    *values;

/* Values allocated by a CPL function (with the xmemory model) */
values = cpl_image_get_values(obj);
...

/* Values deallocated (with the xmemory model: free is overloaded) */
free(values);
```

This solution has the following limitation: if you are using an external C library (other than the CPL) that allocates some memory for you, you have to make sure not to use the `xmemory` functions to deallocate it. In this case, you may want to go for the Solution 2.

Problem 2 not solved by Solution 1:

```
#include "xmemory.h"
...
cpl_image *obj;
ext_image *ext_obj;
double    *values;
double    *ext_values;

/* Values allocated by a CPL function (with the xmemory model) */
values = cpl_image_get_values(obj);

/* ext_values allocated by an ext. lib. (without the xmemory model) */
ext_values = ext_lib_get_values(ext_objs);
...

/* Values deallocated (with the xmemory model: free is overloaded) */
free(values);

/* ext_values deallocated (with the xmemory model: free is overloaded) */
free(ext_values);               /* ERROR! */
```

Solution 2: This solution is to be used in cases where you have to distinguish the data allocated in external libraries from the rest. In this case, you can avoid the standard memory function overloading by including the *Common Pipeline Library* memory handling interface defined in `cpl_memory.h`, and using the functions `cpl_malloc()`, `cpl_realloc()`, `cpl_calloc()` and `cpl_free()` (using the `xmemory` model) everywhere in your code, reserving the functions `free()`, `malloc()`, etc., for handling standard allocated memory coming from these external libraries.

Problem 2 solved by Solution 2:

```
/* #include "xmemory.h" */
#include "cpl_memory.h"
...

cpl_image *obj;
```

```
ext_image *ext_obj;
double    *values;
double    *ext_values;

/* Values allocated by a CPL function (with the xmemory model) */
values = cpl_image_get_values(obj);

/* ext_values allocated by an ext. lib. (without the xmemory model) */
ext_values = ext_lib_get_values(ext_obj);
...

/* Values deallocated (with the xmemory model : through the CPL call) */
cpl_free(values);

/* ext_values deallocated (without the xmemory model) */
free(ext_values);
```

This solution would allow you to deallocate both kind of allocated objects.

It can still happen that CPL objects are built with the ad-hoc constructor and an external allocated array. This would result in a CPL object for whom a part has been allocated with the memory model and another part with the default allocation functions.

Problem 3:

```
#include "cpl_memory.h"
...

ext_image  *ext_obj;
double     *ext_values;
cpl_matrix *m;

/* ext_values allocated by an ext. lib. (without the xmemory model) */
ext_values = ext_lib_get_values(ext_objs);

/* A CPL matrix is built using this array (with the memory model) */
m = cpl_matrix_new_from_data(ext_values, 5, 5);
...

/* Delete the matrix with the CPL destructor (with the memory model) */
cpl_matrix_delete(m);     /* ERROR! */
```

Solution 3: In this particular case, a special CPL object destructor must be used. This must have the ability to destroy only the object but not the data it contains. These data must be deallocated without using the memory model (in the same way that they have been created).

Problem 3 solved by Solution 3:

```
#include "cpl_memory.h"
...
```

```
ext_image  *ext_obj;
double     *ext_values;
cpl_matrix *m;

/* ext_values allocated by an ext. lib. (without the xmemory model) */
ext_values = ext_lib_get_values(ext_objs);

/* A CPL matrix is built using this array (with the memory model) */
m = cpl_matrix_new_from_data(ext_values, 5, 5);
...

/* Delete the matrix not the data (with the memory model) */
cpl_matrix_delete_but_data(m);

/* Delete the data (without the xmemory model) */
free(ext_values);
```

B The PDRM source code

This appendix provides the complete source of the PDRM example discussed in 3.5.

```
#include <cpl_memory.h>
#include <cpl_recipe.h>
#include <cpl_plugininfo.h>
#include <cpl_parameter.h>
#include <cpl_parlist.h>


/* For the my_image_arithmetics prototype */

#include "my_image_arithmetics.h"


#define MY_PLUGIN_VERSION  1


/*
 * Plugin detailed description
 */

static const char *
myplugin_help = "The plugin adds, subtracts, multiplies or divides "
                "two images depending on the operation choosen by the "
                "parameter ‘operation’.";


/*
 * Forward declarations of the initalization, execute and
 * cleanup handlers
 */

int myplugin_create(cpl_plugin *);
int myplugin_exec(cpl_plugin *);
int myplugin_destroy(cpl_plugin *);


int
cpl_plugin_get_info(cpl_pluginlist *list)
{

    cpl_recipe *recipe = cpl_calloc(1, sizeof *recipe);
    cpl_plugin *plugin = (cpl_plugin *)recipe;


    cpl_plugin_init(plugin,
                    CPL_PLUGIN_API,
                    MY_PLUGIN_VERSION,
                    "myplugin",
```

```
                    "Do basic arithmetics on two images",
                    myplugin_help,
                    "Gill Bates",
                    "gbates@macrohard.com",
                    "GPL",
                    myplugin_create,
                    myplugin_exec,
                    myplugin_destroy);

    cpl_plugin_list_append(list, plugin);

    return 0;

}


static int
myplugin_create(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;
    cpl_parameter *p;


    recipe->parameters = cpl_parlist_new();

    p = cpl_parameter_enum_new("myplugin.operation",
                               CPL_TYPE_STRING,
                               "Arithmetic operation to apply.",
                               "myplugin",
                               "add", 4,
                               "add", "subtract", "multiply", "divide");
    cpl_parameter_set_alias(p, "op", NULL, NULL);
    cpl_parlist_append(recipe->parameters, p);

    return 0;

}


static int
myplugin_exec(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;

    return my_image_arithmetics(recipe->parameters, recipe->frames);

}
```

```
static int
myplugin_destroy(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;


    cpl_parlist_delete(recipe->parameters);

    return 0;

}
```

C Comment conventions

Each file in the library begins with a header containing information about the file, such as the file version, the file author, what is contained in the file, etc..

Here is a template of what is put at the head of each `.c` source file in the library:

```
/* $Id: conventions.tex,v 1.17 2003/12/15 16:03:06 dmckay Exp $
 *
 *   This file is part of the ESO Common Pipeline Library
 *   Copyright (C) 2001-2003 European Southern Observatory
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/*
 * $Author: dmckay $
 * $Date: 2003/12/15 16:03:06 $
 * $Revision: 1.17 $
 * $Name:  $
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include ...
#define  ...

/**
 * @defgroup <grouptag> <module name>
 *
 * [Module description]
 *
 */

/**@{*/
/* The function code is placed here */
/**@}*/
```

Here is a template that should be filled and put at the head of each `.h` source file in the library:

```
/* $Id: conventions.tex,v 1.17 2003/12/15 16:03:06 dmckay Exp $
 *
 *   This file is part of the ESO Common Pipeline Library
 *   Copyright (C) 2001-2003 European Southern Observatory
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/*
 * $Author: dmckay $
 * $Date: 2003/12/15 16:03:06 $
 * $Revision: 1.17 $
 * $Name:  $
 */

#ifndef TEMPLATE_H
#define TEMPLATE_H

#include <cpl_macros.h>
#include ...
#define  ...

CPL_BEGIN_DECLS
/* The function declarations are placed here */
CPL_END_DECLS

#endif /* TEMPLATE_H */
```

The fields *Id*, *Author*, *Date* and *Revision* are automatically filled by the configuration control system *CVS*.

The functions are themselves documented using the following template that has to be filled and put just before the function:

```
/*----------------------------------------------------------------*/
/**
  @brief
```

```
  @param
  @param
  @return
 */
/*----------------------------------------------------------------*/
```

Online documentation may then be generated using *doxygen*.

The functions must be documented in the `.c` file. Function documentation must contain information about the function interface (how to call it, what to expect, where to use it, ...) and information about how the function has been written (algorithm used, has it been optimised, ...).

As an example, here is a very simple `.h` file, which illustrates the conventions described above.

```
/* $Id: conventions.tex,v 1.17 2003/12/15 16:03:06 dmckay Exp $
 *
 *   This file is part of the ESO Common Pipeline Library
 *   Copyright (C) 2001-2003 European Southern Observatory
 *
 *   This program is free software; you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation; either version 2 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program; if not, write to the Free Software
 *   Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

/*
 * $Author: dmckay $
 * $Date: 2003/12/15 16:03:06 $
 * $Revision: 1.17 $
 * $Name:  $
 */

#ifndef CPL_IMAGE_H
#define CPL_IMAGE_H

/*-----------------------------------------------------------------------------
                                Includes
 ----------------------------------------------------------------------------*/

#include <stdio.h>
#include <string.h>
#include <cpl_macros.h>
```

```
CPL_BEGIN_DECLS

/*----------------------------------------------------------------------------*/
/**
  @brief    The cpl image structure.
 */
/*----------------------------------------------------------------------------*/
typedef struct _cpl_image_ {
    /* Size of the image in x and y */
    int                 nx, ny ;
    /* Type of the pixels used for the cpl_image */
    cpl_type            type ;
    /* Pointer to pixel buffer as a 1d buffer */
    void            *   pixels ;
    cpl_sparseimage *   badpixelmap ;
} cpl_image ;


/*----------------------------------------------------------------------------
                            Function prototypes
 ----------------------------------------------------------------------------*/

/* Image constructors */
cpl_image * cpl_image_new_empty(void) ;
cpl_image * cpl_image_new_double(const int, const int) ;
cpl_image * cpl_image_new_float(const int, const int) ;
cpl_image * cpl_image_new_int(const int, const int) ;
cpl_image * cpl_image_new_bin(const int, const int) ;
cpl_image * cpl_image_load_float(const char *, const int, const int) ;
cpl_image * cpl_image_load_double(const char *, const int, const int) ;
cpl_image * cpl_image_load_int(const char *, const int, const int) ;

...


CPL_END_DECLS

#endif
/* end of cpl_image.h */
```