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 5.3.1

Date 2011–03–23

	Name	Date	Signature
Prepared:	CPL Project Team	2011-03-23	
Approved:	P. Ballester		
Released:	M. Peron		

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
1.0.1	24/08/2003	All	Corrected errors in cpl_plugin interface examples
2.0.0	01/04/2005	All	Major changes for CPL 2.0 release
2.0.1	14/04/2005	All	Remove obsolete references to CPL 1.0
2.1.0	20/07/2005	All	Update for CPL 2.1
3.0.0	24/08/2006	All	Update for CPL 3.0
4.0.0	27/08/2007	All	Update for CPL 4.0
4.1.0	28/03/2008	All	Update for CPL 4.1
5.0.0	28/04/2009	All	Update for CPL 5.0
5.2.0	11/10/2010	All	Update for CPL 5.2
5.3.1	23/03/2011	All	Update for CPL 5.3.1, include License for CPL based code

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 (images, tables, matrix, vectors, ...) and their associated methods (**libcplcore**).
- Support for dynamic loading of recipe modules and standardised application interfaces for pipeline recipes (**libcplui**).
- Image and signal processing capabilities and standard implementations of commonly used data reduction tasks (**libcpldrs**).
- DFS specific functionalities to insure the DFS compliance of the pipelines (**libcpldfs**).

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

Standardised versions of the most common calibration steps and removal of instrument signature are now offered. Of course the data reduction system developers may still define any specific procedure to support bias subtraction, flat fielding, wavelength calibration, instrument response linearisation, cosmic ray removal, object detection, bad pixel determination, etc., as needed for a particular instrument.

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.

Major areas of growth foreseen for future releases are general astronomical utility functions enabling spherical coordinate transformations, date and time conversions, precession, atmospheric extinction determination and other common operations in astronomy.

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 the past, also M. Kiesgen, and D.J.-McKay made major contributions to CPL.
Currently, the CPL project team consists of: K. Banse, S. Castro, C. Izzo, L. de Bilbao, L. Lundin.

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 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 start 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:

- Linux (glibc 2.1 or later)
- Mac OSX 10.0 or later
- BSD compatibles

However, only the VLT target platforms and operating systems, Scientific Linux 5.x and Linux (glibc 2.1 or later), are officially supported, right now.

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:

- 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
- The GNU `make` utility
- If you want to use **gasgano**, also the Java SDK (Software Develpment Kit) from Sun

To actually use the *Common Pipeline Library* you need:

- The **CFITSIO** FITS utility library from NASA
- The **WCSLIB** library from Mark Calabretta (http://www.atnf.csiro.au/ mcalabre/WCS)
- The **FFTW** library for FFT calculations (http://www.fftw.org)

CPL uses **CFITSIO** as FITS I/O library.
The CPL library is synchronized with the ESO VLT-software which uses CFITSIO rel. 3.0.9, right now. Thus, CPL also uses CFITSIO 3.0.9, and is tested right now for this specific CFITSIO version. Support will be for the CFITSIO versions 3.x.y in the future. Whenever, ESO's VLT software project upgrades to a newer version of CFITSIO, then CPL will follow.

The CPL functions related to the World Coord. System are based on the WCSLIB code. Currently, CPL uses WCSLIB 4.4.4 - higher versions of WCSLIB may work as well with CPL, depending about the backwards compatibility of this higher version.
Fast Fourier transforms in CPL functions rely on the FFTW library. The version 3.1.2 of FFTW is currently used in CPL. Again, higher versions of FFTW may work as well with CPL, depending about the backwards compatibility of this higher version.

Please, note that FFTW and WCSLIB functions are used only by a few CPL functions, so your pipeline code may well work without these additional libraries. However, when installing several CPL based pipelines, chances are much higher that one pipeline code might need also WCSLIB or FFTW.

2.2.2 Downloading the CPL source distribution

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

```
http://www.eso.org/sci/data-processing/software/cpl/donwload.html
```

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, since *Common Pipeline Library* depends on release 3.0.9 of the **CFITSIO** library (see section 2.2.1), this specific version of **CFITSIO** is also available from the official ESO-CPL download page as specified above.

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 **CFITSIO** (c.f. section 2.2.1) does not already exist on your system, compile and install the **CFITSIO** library. For detailed instructions on how to install the **CFITSIO** library, please, refer to the **CFITSIO** documentation.

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

```
$ zcat -d CFITSIO.tar.gz | tar -xvf -
$ cd cfitsio
$ ./configure --prefix=/usr/local
$ make
$ make shared
$ make install
```

The following assumes that **CFITSIO** is installed in `/usr/local`.

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

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

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 `CFITSIODIR` tells the configuration script where the **CFITSIO** libraries and header files can be found. Actually, this variable needs to be defined only if **CFITSIO** has not been installed in the default directory `/usr/local` 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 `CFITSIODIR` 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 `CFITSIODIR` 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:

```
CFITSIODIR=/usr/local
CPLDIR=/usr/local
LD_LIBRARY_PATH=$CPLDIR/lib:$CFITSIODIR/lib:$LD_LIBRARY_PATH
export CPLDIR CFITSIODIR 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 CFITSIODIR /usr/local
setenv CPLDIR /usr/local
setenv LD_LIBRARY_PATH \
       $CPLDIR/lib:$CFITSIODIR/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 step 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 `CFITSIODIR` 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 --prefix=/usr/local
$ 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 `$CPLDIR/share/doc/cpl/html`. The on-line documentation for `libcext`, the C Extension Library, which is used inside CPL, can be found in `$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 `CFITSIODIR` are overridden by the options `--prefix` and `--with-CFITSIO` 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. Section 3.6 will describe the procedure to follow in case you want to develop an ESO's VLT instrument pipeline.

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

Licenses

All CPL based development code shall use the GNU Public License, and contain the following text (also included in the standard header of every CPL module):

> 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., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA

Build Tools

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 pipeline template directory tree is available on the CPL web page.

3.3 Linking your application with the CPL

The CPL libraries *libcpldfs*, *libcpldrs*, *libcplui* and *libcplcore*, together with *libcext* and the *libcfitsio* 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.

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$CPLDIR/lib -lcpldfs -lcpldrs -lcplui -lcplcore -lcext \
> -L$CFITSIODIR -lcfitsio ...
```

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 *libcfitsio* 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 -lcpldfs -lcpldrs -lcplui -lcplcore ...
```

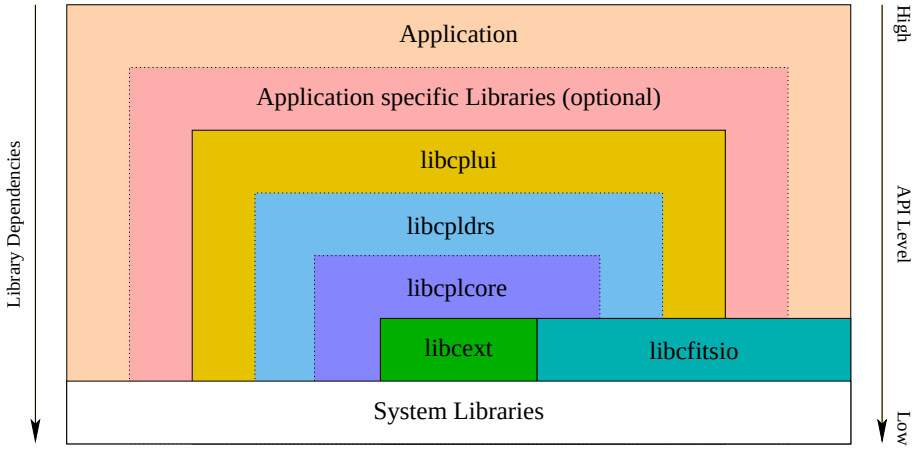


Figure 1: Library dependencies of a CPL application

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$CPLDIR/lib -lcplcore -lcext -L$CFITSIODIR -lcfitsio ...
```

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.h>

int main()
{

    cpl_init(CPL_INIT_DEFAULT);

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

    cpl_end();

    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.h>
```

includes the prototype of all the CPL functions. You must include this file wherever you are using any CPL function.

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(CPL_INIT_DEFAULT);
```

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. 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_end();
```

shuts down the CPL 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 5.2.10 and the CPL reference manual [1].

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.h>

#define MY_PLUGIN_VERSION  1

/* Plugin detailed description */

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


static int myplugin_create(cpl_plugin *);
static int myplugin_exec(cpl_plugin *);
static 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,
                    CPL_PLUGIN_TYPE_RECIPE,
                    "myplugin",
                    "Do basic arithmetic on two images",
                    myplugin_help,
                    "Gill Bates",
                    "gbates@macrohard.com",
                    "GPL",
                    myplugin_create,
                    myplugin_exec,
                    myplugin_destroy);

    cpl_pluginlist_append(list, plugin);

    return 0;

}
```

The first line includes all the definitions of the CPL.

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_parameterlist_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, CPL_PARAMETER_MODE_CLI, "op");
    cpl_parameterlist_append(recipe->parameters, p);
```

```
    return 0;

}


static int
myplugin_destroy(cpl_plugin *plugin)
{

    cpl_recipe *recipe = (cpl_recipe *)plugin;


    cpl_parameterlist_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 [1] 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 A. To try it, you should build a shared object library from the source and you must provide the actual processing function.

3.6 A specific *Common Pipeline Library* application : the VLT instrument pipeline

A VLT instrument pipeline is a very specific CPL-based application. Because of the big number of different pipelines it needs to maintain and develop, ESO imposes on those a series of standards and/or constraints that must be strictly followed:

- The coding style must follow a series of common rules (the error checking must be done extensively, the code must be well documented using the same doxygen documentation tags, etc.).
- The pipeline source directory tree structure must follow the standard (organisation, usage of the GNU tools *autoconf* and *automake* in a standard way, etc.).
- The FITS header keywords access must be done in a standard way.
- The DFS-related parts must be defined in a standard place.
- The **libcext** library must not be used.
- The **CFITSIO** functions may be used directly if they are compatible with the *officially supported* CFITSIO version, currently rel. 3.0.9; mixing different CFITSIO versions may produce unexpected behaviour - no support from ESO can be expected in that case.
- The pipeline products must be written with the proper format, keywords, etc. The information about their existence must be given to `esorex` for further processing.

These are only the main constraints that need to be followed by a VLT instrument pipeline. The total list can be very long, and difficult to describe in a document (especially when it comes to error handling or coding style).

If you want to know more about these specifications, see the DFS Deliverables Specification document [2] and the Data Flow Pipeline and Quality Control Users Manual document [3]. See the ESO DICB – Data Interface Control Document [4] for informations about FITS header keywords.

Of course, if you want to develop your own CPL-based application that is not a pipeline, you still can use the pipeline template and benefit from the fact that the plugin is already properly defined and ready to be executed with *esorex*.

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_load(...);
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, images lists, masks, tables, vectors, ...) and purely functional components to help programmers, such as the messaging and the error handling components.

'Data hiding' is used everywhere. All 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 The extended memory model

The library offers a set of memory allocation/deallocation functions:

```
cpl_malloc()
cpl_calloc()
cpl_realloc()
cpl_free()
cpl_strdup()
```

These functions are meant to replace the default standard library functions that control and handle all memory allocation in applications. The behaviour of these functions is controlled with the configuration of the CPL. By default, they use the standard system memory handling functions. Nevertheless, it is possible to switch on (–enable-memory-mode option of configure) the extended memory functions described here.

4.3.1 Advantages of using the extended memory functions

By using these functions, some information about the allocated and deallocated pointers is internally kept. This way, the system knows at any moment the list/size of the still allocated pointers, making it easier to track memory leaks.

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

```
cpl_memory_is_empty()
cpl_memory_dump()
```

4.3.2 Drawbacks of using the extended memory functions

These functions keep internally various informations on every single pointer that is currently allocated. Thus, you need to know when you install CPL which value you are never going to exceed in terms of number of pointers allocated at the same time in your programs. The default is currently set to 200000, which should be enough for most applications.

Note that the cpl_propertylist uses a lot of pointers when it contains large FITS headers. In order not to exceed this limit, you may try not to load all your input files headers in property lists at the same time if this is not necessary.

If the maximum number of pointers your application may need is bigger than that (say around 300000), you need to specify this when you install CPL:

Instead of typing:

```
$ ./configure --enable-memory-mode=2
```

you may type:

```
$ ./configure --enable-max-ptrs=500000 --enable-memory-mode=2
```

To increase the supported number of pointers, you just need to reconfigure, recompile and reinstall CPL.

Note that the pointers information table is statically allocated, and that using enormous values (i.e. table size) would cause the memory consumption of CPL unreasonably high.

4.3.3 Using the extended memory

The memory allocated inside the CPL has to be deallocated using the provided memory handling 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 the CPL memory functions to allocate/deallocate your memory in your code with `cpl_malloc()`, `cpl_calloc()`, `cpl_realloc()`, `cpl_strdup()` or `cpl_free()`.

The only rule is that all the memory allocated with the CPL memory functions must be deallocated with them.

If you do not want to use the extended memory system in your application, and do not want it to be used in CPL, you can configure CPL with the option –enable-memory-mode=0 like this:

```
$ ./configure --enable-memory-mode=0
```

This way, the offered functions `cpl_malloc()`, `cpl_calloc()`, `cpl_realloc()`, `cpl_strdup()` or `cpl_free()` will simply call the associated system functions.

This is the default behaviour from CPL version 4.0 on.

4.4 Error handling

Error handling in the CPL is done through the `cpl_error` component (see Section 5.2.11).

4.5 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.

4.6 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.

4.6.1 Objects

An object is a C *structure* that contains all the information needed to describe it. The objects included in 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 as an array of pixels, its image size in X and Y, its pixel type and possibly a bad pixel map; nothing more.

If more complicated objects are needed, it is left to the developer to define higher level objects based on the CPL objects and other opportune parameters and attributes.

Each object has one *constructor* which allocates the necessary memory, and a *destructor* to deallocate it. The destruction of objects should always be done through its dedicated method.

4.6.2 Methods

Apart from the constructor and destructor, each function which operates on an object is called 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 an object without modifying it.

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

4.6.3 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 appropriate error code otherwise) or by returning a conventional value (such as a NULL pointer when a valid pointer is expected) and setting appropriately the error code (see section 5.2.11).

4.7 Naming Conventions

The following defines the construction of a CPL function name - and other types of identifiers in the CPL namespace. This enables uniformity of nomenclature, easing the search and the identification of either known or unknown functionality.

Symbolic constants shall conform to the following naming conventions:

- A CPL symbolic constant name consists of fields, which are separated by the the underscore character (_).
- A field starts with an upper-case letter and is followed by upper-case letters and digits.
- The first field shall be `CPL`.

A CPL function name adheres to the following rules of syntax:

- A CPL function name consists of fields, which are separated by the the underscore character (_).
- A field starts with a lower-case letter and is followed by lower-case letters and digits.
- Each field has a meaning, depending on its place in the sequence of fields.

A CPL macro (that is a `#define` accepting arguments, as opposed to symbolic constants) shall conform to the same naming conventions as those of CPL functions.

4.7.1 Meaning of Fields

The different fields composing a CPL function name have a specific meaning, depending on their position in the name.

The fields are:

Library The first field is the library name, i.e., the library to which the function belongs. In the case of the CPL functions, is it always set to `cpl`. The library field is mandatory.

Subject The subject refers to the main CPL section where the function is defined and implemented. It may be one of several subject types:

1. Object - This is the CPL object which is handled by the function. Objects are, for instance, `image` and `table`.
2. Domain - This is the functional area in which the function has been inserted. The domain is used to group functions sharing similar scope, but not acting on a specific object, such as the CPL messaging, `geom` (geometry), and `photom` (photometry) functions.
3. Exceptions - There are a few CPL functions which do operate neither on a CPL object nor within a given domain. Exceptionally, the names of these functions do not include any subject field: `cpl_{init,free,assure}()`, `cpl_{malloc,realloc,calloc,strdup}()`.

The subject field is mandatory, with the mentioned exceptions.

Verb The verb defines the action on the subject.

1. Existentials - These indicate the creation or destruction of an object (textttnew, `delete`, `create`...). For instance, the function named `cpl_vector_new()` creates a new CPL vector.
2. Morphologicals - Change the size of an existing CPL object. The number of elements within the object is changed. For instance, `cpl_matrix_append()` is used to append a matrix to another one whose size is therefore modified.
3. Elementwise operators - These are functions that act on each specified element of the CPL object. For instance, `cpl_image_add()` sums two images, pixel by pixel.
4. Global operators - These are functions that act on a CPL object as a whole. For instance, `cpl_vector_corre` is used to correlate two vectors.
5. Generic - The following do not fall into the above categories: `get`, `set`, `is`, `has` and `dump`.

6. Exceptions - These are verbs which do not apply to neither a CPL object nor a domain. Strictly speaking, they are subject free. See the above exceptions.

The verb field is mandatory.

Qualifier A qualifier specifes the object or concept upon which the verb acts, in the context of the subject. There are three types of qualifiers:

1. Read/Write Attributes - These are attributes of a CPL object that may be set or retrieved, as in the functions `cpl_polynomial_{set,get}_coeff()`.
2. Read-only Attributes - These are attributes of a CPL object for which it is not meaningful to set a value, although the object possesses one that may be computed, as in the function `cpl_image_get_median`
3. Others E.g. `cpl_table_new_from_model()`.

The qualifier field is optional.

Item/Concept A further specification of the functionality, e.g.

1. CPL object(s)
2. attributes of an existing CPL object (e.g. `size`).
3. primitive *C* type (`string` for `char *` and `int`, `float`, `double`)

This field is optional.

Sub-item A further specification of the functionality, e.g.

1. attributes of an existing CPL object (`column`, `row` and `window`).
2. primitive *C* type (`string` for `char *` and `int`, `float`, `double`)

This field is optional.

4.7.2 Lexicon

Subject The following words are permitted as subjects. These represent the modules in CPL. Objects that end with set or list are collective objects, while the rest are singular objects. list indicates that the collection of objects is ordered, while set indicates that the collection is unordered. (Strictly speaking, `table` is thus a `column`-set).

- `array`
- `bivector`
- `column` — Internal to CPL
- `error`
- `frame`
- `frameset`
- `image`

- `imagelist`
- `mask`
- `matrix`
- `memory`
- `msg`
- `parameter`
- `parameterlist`
- `plugin`
- `pluginlist`
- `polynomial`
- `property`
- `propertylist`
- `stats`
- `table`
- `tools` — Internal to CPL
- `type`
- `vector`

The following words are also permitted as subjects. These represent functional areas in the higher-level sections of the CPL. This list will likely be extended.

- `apertures`
- `detector`
- `dfs`
- `fit`
- `flux`
- `geom`
- `photom`
- `ppm`
- `wcs`

Verb The following words are permitted as verbs.

Existentials

- `delete` (Destructor)
- `unwrap` (Destroys object, leaving internal components intact, c.f. `wrap`)
- `wrap` (Constructor of a new object composed around existing data, c.f. `unwrap`)
- `duplicate` (Copy constructor)

- `extract` (Create a new object which contains a part of another object, e.g. create a vector from part of another vector)
- `load` (Constructor from file, c.f. `save`)
- `new` (Constructor)
- `save` (Create file of an existing object, c.f. `load`)
- `offset` (Image combination in `cpldrs`)
- `filter` (Filtering always create a new object)
- `cast` (Casting always create a new object)

Morphologicals

- `append` (Add an element to the tail of an object)
- `collapse` (Remove a dimension of a multi-dimensional object)
- `erase` (Remove element(s) from an object, c.f. `insert`)
- `insert` (Add an element to an object, c.f. `erase`)
- `prepend` (Add an element to the head of an object)

Global operators

- `correlate` (Compute the cross-correlation between two objects)
- `count` (Get the number of occurrences of some object attribute)
- `shift` (Rearrange elements in a CPL object)
- `find` (Locate an element within a CPL collective object)
- `interpolate` (Compute an interpolated value)
- `flip` (Reverse the order of elements in a CPL object)
- `fft` (Compute the FFT of a CPL object)
- `turn` (Rotate the elements of a CPL object through a multiple of 90 degrees)
- `invert` (Compute the inverse (matrix))

Elementwise operators

- `abs` (Absolute value of each element)
- `add` (Add elements at equivalent positions)
- `and` (Binary AND on elements at equivalent positions)
- `average` (Determine the average of elements at equivalent positions)
- `cast` (Convert the type of elements in an existing object)
- `copy` (Overwrite some/all elements in an existing object)
- `divide` (Divide elements at equivalent positions)
- `fill` (Assign values to specified elements within a CPL object)
- `reject` (Flag element, e.g. set bad-pixel, c.f. `accept`)
- `labelise` (Assign numeric labels to associated elements)
- `multiply` (Multiply elements at equivalent positions)
- `normalise` (Rescale elements to lie within a given range)
- `not` (Binary NOT on elements)
- `or` (Binary OR on elements at equivalent positions)

- `subtract` (Subtract elements at equivalent positions)
- `threshold` (Assign a value to elements whose value lies outside a specified range)
- `accept` (Unflag element, e.g. remove bad-pixel, c.f. `reject`)
- `xor` (Binary XOR on elements at equivalent positions)

Generic

- `get` (Retrieve the value of an attribute associated with an object)
- `set` (Assign a value to an attribute associated with an object)
- `dump` (Print the object content to stream, for debugging)
- `is` (Used for checking existence or state)
- `has` (Used for checking existence or state)

Additionally, the following words are permitted as verbs in subject free function names.

- `assure` (Ensure the presence of a given condition and handle the case where this is not true)
- `calloc` (Allocate memory initiliased to zero)
- `free` (Deallocate memory associated with a pointer)
- `init` (Initialise an object and system)
- `malloc` (Allocate memory)
- `realloc` (Reallocate the memory associated with a pointer)
- `strdup` (Duplicate a character array)

Qualifiers The words permitted as qualifiers are listed in appendix C.

Items The words permitted as items are listed in appendix C.

Sub-items The following words are permitted as sub-items:

- `column`
- `double`
- `float`
- `int`
- `row`
- `string`
- `window`

5 The CPL components

5.1 Component libraries

The functionality of the CPL is provided by four component libraries, implementing the low-, medium- and high-level CPL interfaces respectively, plus a DFS specific functions library. 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.

The **libcplui** library implements the medium-level data types and utilities serving as an interface to the pipeline run-time environment.

Standard implementations for instrument-independent data-reduction functions and functions for monitoring the data quality are provided by the **libcpldrs** library.

Finally, the **libcpldfs** library is there to insure the compliance of the pipeline products by implementing some of the important DFS requirements on the pipeline products.

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.

For access to FITS data files, the CPL internally relies on the **CFITSIO** FITS I/O library. Since the CPL provides high-level facilities to read and write data from/to a FITS file, direct calling of **CFITSIO** functions is only permitted on exceptional cases, and within the scope of *ad hoc* loading and saving functions.

The low-level library *libcext*, delivered together with the CPL, is an internal library exclusively used by the CPL and its functions shall not be called directly by any VLT/VLTI pipeline.

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 *complex*). 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 whose name start with *cpl_image* deal with images. Some of them return a newly allocated image (*cpl_image_xxx_create()*, *xxx_new()*, *xxx_wrap_xxx()* or *xxx_load()* functions) and some others work locally on the passed image. The newly allocated images must later be deallocated with one of the destructors (`cpl_image_delete()` or `cpl_image_unwrap()`).

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
- basic image operations, normalisation, thresholding, averaging, collapsing, extraction or flipping
- various statistical computations on images
- linear, median or morphological filtering operations
- resampling functionalities
- 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 (`cpl_image` and `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 *cannot* be accessed directly; every developer must/can 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**

   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 complex 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. 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_mask        *   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

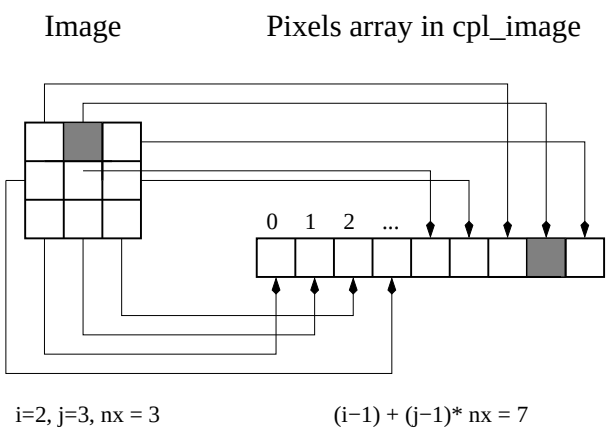


Figure 2: Pixel storage in the 1D data array

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).

These fields *cannot* be accessed directly. They are shown here for information pupose. Accessor functions are provided to access the pixels or the image informations (see IO routines description).

2. **The image IO routines**

   There are four kind of functions that can be used to generate *cpl_image* objects.

   The `cpl_image_new()` function will create a new image of the specified size and type, with pixels values set to 0 and an empty bad pixel map.

   The `cpl_image_load()` function 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*.

   The `cpl_image_wrap_xxx()` functions will create a *cpl_image* object around an already existing passed data array. This image will have to be deallocated with the *cpl_image_unwrap()* function.

   The `cpl_image_new_from_xxx()` functions will create newly allocated images using data coming from other CPL objects.

   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(1024, 512, CPL_TYPE_FLOAT);
```

```
/* 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(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.

3. **The basic image operations**

   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.

   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 3.

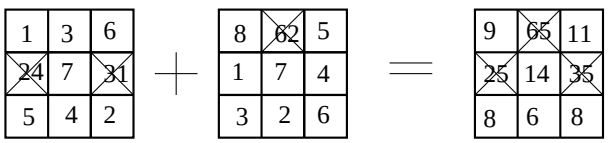


Figure 3: Bad pixel map handling in basic images operations

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 4).

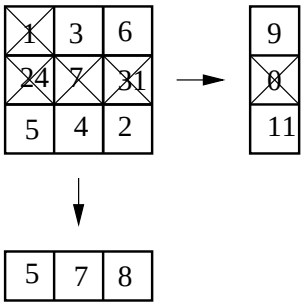


Figure 4: Bad pixel map handling in the collapsing function

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

4. **Statistics on images**

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 5.

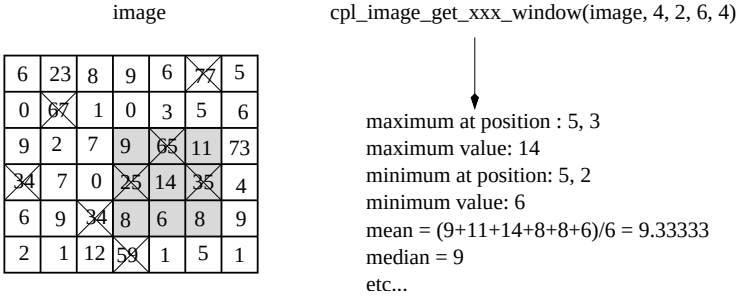


Figure 5: Bad pixel map handling in statistics computations

5. **The image filtering functions**

   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 6.

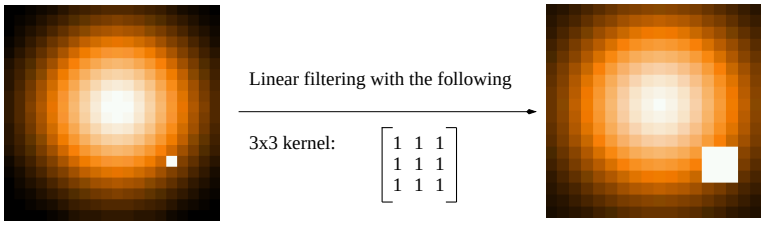


Figure 6: Filtering without bad pixels handling

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

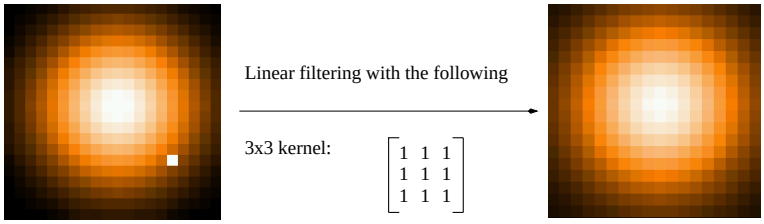


Figure 7: Filtering with the pixel (16, 6) tagged as bad

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 as bad pixels in the filtered image.

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

5.2.2 Masks

A *cpl_mask* is a two dimensions map in which the elements can only have two different values. This object is used to represent bad pixel maps or binary images.

Binary images are widely used (and very useful) in image processing for object or edge detection.

This object 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 (*cpl_mask_threshold_image_create()*) to "binarise" an image is provided. Figure 8 illustrates its effect on an example, where the threshold is computed with the *cpl_image_stats* functions on the input image to obtain a *cpl_mask* object.

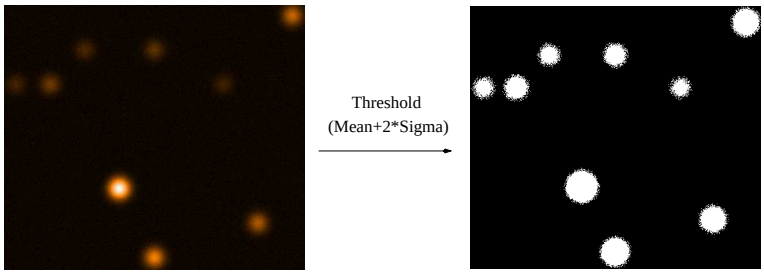


Figure 8: Use of thresholding to binarise an image to a mask

Some simple morphological operation can be applied to the mask to make one connected object out of each detected star as shown in Figure 9. The operation applied here is a closing (erosion + dilation).

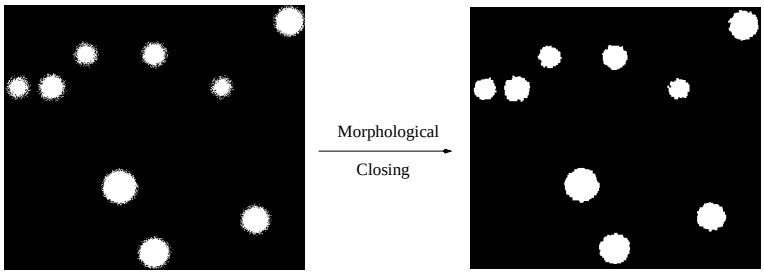


Figure 9: Effect of a morphological closing

Once the different objects are connected, we can apply a labelisation (with *cpl_image_labelise_mask_create()*) on the mask to differentiate them automatically (see Figure 10). The mask is transformed into an integer image where the non-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 like it is done in the section 5.4.1.

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

5.2.3 List of images

The *cpl_imagelist* object is an extension of the *cpl_image* object. It is a container for several images. A list of images can only contain images of the same type, and of the same size. To ensure the validity of an image list (basically that these conditions are verified), one can use the *cpl_imagelist_is_uniform()* function.

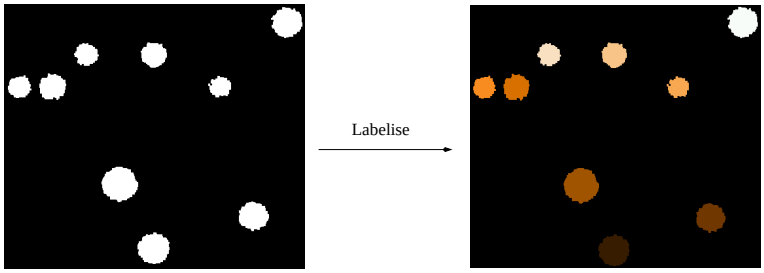


Figure 10: Labelisation of a mask to an integer image

The two main ways to create an image list are either to load one from a FITS file extension with *cpl_imagelist_load()* or from a set of frames with *cpl_imagelist_load_frameset()*, or to create one ”by hand” with calls to *cpl_imagelist_new()* and *cpl_imagelist_set()*.

Every image list must be deallocated using *cpl_imagelist_delete()*. Note that if you set images in an image list, you have to leave those images allocated, they will be deallocated by the *cpl_imagelist_delete()* call.

Once you have your image list created, you can perform series of simple operations between an image list and an image, or a scalar. You also can collapse an image list, normalise it or threshold it.

5.2.4 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 strictly based on this definition.

Currently, five basic numerical types are supported for a CPL table column: `CPL_TYPE_INT`, `CPL_TYPE_FLOAT`, `CPL_TYPE_DOUBLE`, `CPL_TYPE_FLOAT_COMPLEX`, and `CPL_TYPE_DOUBLE_COMPLEX`. A type indicating columns made of character strings, `CPL_TYPE_STRING`, is also supported. From the above mentioned basic types, array types can be derived, *i.e.*, a table column element may be an array of numbers, or an array of character strings.

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. It is possible to flag each *cpl_table* row as ‘selected’ or ‘unselected’, and each column's element as ‘valid’ or ‘invalid’. 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.h>
...
int main()
{
    ...
    cpl_table *table;
    int        number_of_rows;
    int        depth;
    ...

    cpl_init(CPL_INIT_DEFAULT);
    ...
    number_of_rows = 100;
    depth = 5;
    ...
    table = cpl_table_new(number_of_rows);

    cpl_table_new_column(table, "Player", CPL_TYPE_STRING);
    cpl_table_new_column(table, "Games won", CPL_TYPE_INT);
    cpl_table_new_column(table, "Games lost", CPL_TYPE_INT);
    cpl_table_new_column(table, "Success rate", CPL_TYPE_FLOAT);
    cpl_table_new_column_array(table, "Scores", CPL_TYPE_INT, depth);
    cpl_table_new_column_array(table, "Other players", CPL_TYPE_STRING, depth);
    ...
    cpl_table_delete(table);
    ...
    cpl_end();
    return 0;
}
```

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

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

    cpl_init(CPL_INIT_DEFAULT);
    ...

    /*
     *  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_2005.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_2005.fits", 0);
    cpl_table_delete(table);
    ...
    cpl_end();
    return 0;
}
```

It is also possible to load part of a FITS table into memory: this may turn advantageous in case of very large tables. This can be done using the function `cpl_table_load_window()` instead of `cpl_table_load()`. For instance, in order to load 4 rows starting from row 2, in the above example the call to `cpl_table_load()` should be replaced by

```
table = cpl_table_load_window("Championship_2005.fits", 2, 1, NULL, 2, 4);
```

The fourth argument of this function may also be used, for defining a subset of columns to be loaded.

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

- Defining and allocating new columns.
- Creating new columns pointing to external data.
- Reading and writing table cells.
- Shifting positions of column values.
- Supporting invalid table cells, and invalid array elements.
- 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.
- 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* [1] but, in the following, some of the functionalities are explained with the help of a number of simple examples.

1. **Accessing table elements**

   A table column can be accessed by specifying its name, while one of its elements can be accessed by specifying its table row number. As mentioned above, a table column may also consist of arrays of the basic supported types. In this case by specifying a column name and a table row number an array will be returned, whose elements will then be accessed by specifying their position along the array.

   Note that, in the same way as all the columns of a table must have the same length (corresponding to the number of rows in the table), all the arrays in a given column must have the same size. The length of the arrays belonging to the same column is conventionally called the *depth* of the column. In the following example it is shown how to access table elements both from simple columns and from columns of arrays (error checking is omitted for clarity):

```
...
#include <cpl.h>
...
int main()
{
    ...
    cpl_table *table;
    cpl_array *array;
    int        number_of_rows = 100;
    int        depth = 5;
    char      *player;
    int        score;
    ...

    cpl_init(CPL_INIT_DEFAULT);

    table = cpl_table_new(number_of_rows);

    cpl_table_new_column(table, "Player", CPL_TYPE_STRING);
    cpl_table_new_column(table, "Games won", CPL_TYPE_INT);
    cpl_table_new_column(table, "Games lost", CPL_TYPE_INT);
    cpl_table_new_column_array(table, "Scores", CPL_TYPE_INT, depth);
    cpl_table_new_column_array(table, "Other players", CPL_TYPE_STRING, depth);
```

```
/*
 * Writing the name "Ren" as a Player at row 42, and the number of
 * games won and lost.
 */

cpl_table_set_string(table, "Player", 42, "Ren");
cpl_table_set_int(table, "Games won", 42, 0);
cpl_table_set_int(table, "Games lost", 42, 5);

/*
 * Now write the scores: an array of as many values as the depth
 * that was declared for the columns to access. In this case the
 * array is filled with 0.
 */

array = cpl_array_new(depth, CPL_TYPE_INT);
cpl_array_fill_window_int(array, 0, depth, 0);
cpl_table_set_array(table, "Scores", 42, array);
cpl_array_delete(array);

/*
 * At the end the array can (and must) be deleted, since it was
 * physically copied to the table. If efficiency reasons make this
 * duplication of an array impracticable, the cpl_table_set_array()
 * call may be replaced by:
 *
 *   cpl_table_get_data_array(table, "Scores")[42] = array;
 *
 * where the created array is directly "plugged" into the appropriate
 * column element. Of course in this case cpl_array_delete(array) must
 * not be used.
 */

/*
 * Now write the players to the column of arrays of character strings:
 */

array = cpl_array_new(5, CPL_TYPE_STRING);
cpl_array_set_string(array, 0, "Stimpy");
cpl_array_set_string(array, 1, "Goofy");
cpl_array_set_string(array, 2, "Micky");
cpl_array_set_string(array, 3, "Donald");
cpl_array_set_string(array, 4, "Pluto");
cpl_table_set_array(table, "Other players", 42, array);
cpl_array_delete(array);

/*
 * Again, the last two calls may be replaced by the more efficient
 *
 *   cpl_table_get_data_array(table, "Other players")[42] = array;
```

```
     *
     * Note that the analogous
     *
     *   cpl_array_get_data_string(array)[0] = "Stimpy";
     *
     * cannot be used in this case, because "Stimpy" is a constant string
     * that cannot be released by the table destructor.
     */

    /*
     * Now access some of the written data:
     */

    player = cpl_table_get_string(table, "Player", 42);
    score = cpl_table_get_int(table, "Games won", 42);
    ...
    array = cpl_table_get_array(table, "Other players", 42);
    player = cpl_array_get_string(array, 2);
    array = cpl_table_get_array(table, "Scores", 42);
    score = cpl_array_get_int(array, 2);

    /*
     * Do not use:
     *
     *   cpl_free(player);
     *   cpl_array_delete(array);
     *
     * The accessors just return a pointer to an internal element, that
     * will be released at table destruction.
     */

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

2. **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.h>
...
int main()
{
    ...
    cpl_table *table;
    int        i;
```

```
int        nrows = 10;
double     mean;
...

cpl_init(CPL_INIT_DEFAULT);

/*
 * 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(table, "Numbers", CPL_TYPE_INT);
for (i = 0; i < nrows; i++)
    cpl_table_set_int(table, "Numbers", i, i + 1);

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

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

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

mean = cpl_table_get_column_mean(table, "Numbers");

/*
 * Now write again some valid values. 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_get_column_mean(table, "Numbers");

/*
 * In the case of a column of arrays, or also of character strings,
 * invalidating an element means to release it from memory:
 */

cpl_table_new_column(table, "Character strings", CPL_TYPE_STRING);

/*
 * Write a character string to table element 5 of column
 * "Character strings". The test string is duplicated:
 */
```

```
cpl_table_set_string(table, "Character strings", 5, "test string");

/*
 * Invalidating this string means to destroy it:
 */

cpl_table_set_invalid(table, "Character strings", 5);

/*
 * The same happens with a column of arrays: here a column of integer
 * arrays of size 12 is created; then one integer array is created,
 * all its elements are set to 5240, and finally the array is inserted
 * at the cells 5 and 6 of the table column. Note that the created array
 * must have exactly 12 elements, according to the declaration of the
 * column.
 */

cpl_table_new_column_array(table, "Arrays of integers", CPL_TYPE_INT, 12);
array = cpl_array_new(12, CPL_TYPE_INT);
cpl_array_fill_window_int(array, 0, 12, 5240);
cpl_table_set_array(table, "Arrays of integers", 5, array);
cpl_table_set_array(table, "Arrays of integers", 6, array);

/*
 * Since the array is physically copied to the table, it can (and it
 * should!) be released:
 */

cpl_array_delete(array);

/*
 * As with the character string column, invalidating a table cell
 * means to destroy the copy of the array:
 */

cpl_table_set_invalid(table, "Arrays of integers", 5);

/*
 * How to invalidate a single array element? Here is shown how to
 * invalidate element 2 of array 6:
 */

array = cpl_table_get_array(table, "Arrays of integers", 6);
if (array)
    cpl_array_set_invalid(array, 2);

/*
 * The array read from the table should not be released, because it
 * belongs to the table itself: cpl_table_get_array() just returns
 * a handle to an internal object. All the memory associated to
 * the table is released when the table is destroyed:
```

```
     */

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

It should be underscored that when any table column value is flagged as *invalid*, it is lost: there is no function to set an invalid element back to its original value. The only way to validate a table element is to write a value to the corresponding position. 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.h>
...
int main()
{
    ...
    cpl_table *table;
    float     *data;
    int        size;
    ...

    cpl_init(CPL_INIT_DEFAULT);


    /*
     * It is here assumed that the float column "Data" contains some
     * invalid values. The data buffer of the table column is extracted
     * and passed to an external fitting routine, but this is a
     * mistake: in fact the buffer 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_fill_invalid_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_erase_invalid() removes from a table
     * any row containing at least one invalid value.
     */

    cpl_table_erase_invalid(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);
    ...

    cpl_end();
    return 0;
}
```

The most obvious example of exporting a column's internal data buffer to an external process is when a table is converted to FITS format and written to disk. This is done by the function `cpl_table_save()`, that converts any invalid column value into 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`, which 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_fill_invalid_int()` for each table column of type `int` containing invalid values, and this should be done 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). Here is a simple example:

```
...
#include <cpl.h>
...
int main()
{
```

```
    ...
    cpl_table *table;
    int        nrows = 10;
    ...


    cpl_init(CPL_INIT_DEFAULT);

    /*
     * 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(table, "Numbers", CPL_TYPE_INT);
    cpl_table_fill_column_window_int(table, "Numbers", 0, nrows, 3);

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

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

    cpl_end();
    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.

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


    cpl_init(CPL_INIT_DEFAULT);

    /*
     * 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);

    cpl_end();
    return 0;
}
```

In this 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 is automatically excluded by the subtraction operation.

Elements shifting is not supported for character string columns and for columns of arrays.

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


    cpl_init(CPL_INIT_DEFAULT);

    /*
     * 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
     * 2006; 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", CPL_EQUAL_TO, 1958);
```

```
    cpl_table_or_select_int(table, "Year", CPL_EQUAL_TO, 2005);
    cpl_table_and_select_string(table, "Month", CPL_EQUAL_TO, "^[Aa].*");
    cpl_table_and_select_float(table, "Day", CPL_NOT_LESS_THAN, 5.5);
    cpl_table_and_select_float(table, "Day", CPL_LESS_THAN, 12.3);
    selected = cpl_table_or_select_string(table, "Month", CPL_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);
    cpl_end();
    return 0;

}
```

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.

In the specific case of complex numbers, only the CPL_EQUAL_TO and CPL_NOT_EQUAL_TO are applicable.

5. **Tables of images**

   As seen above, it is possible to define tables containing columns of arrays. In principle, each array can be viewed as a storage for values that may be cast into more complex data structures – for instance images, cubes, etc.. The concept of *column dimension* has been introduced for this purpose. In the following example it is shown how to create a table containing a column made of 2-dimensional images (error checking is omitted for clarity):

```
...
#include <cpl.h>
...
int main()
{
    ...
    cpl_table *table;
    cpl_array *array;
    cpl_image *image;
    int        rows = 12;            /* Number of images = rows in table  */
    int        naxis = 2;            /* Number of axis of each image      */
```

```
int        size[] = {25, 33};     /* Size of one image: x = 25, y = 33 */
int        depth;
int        i;


cpl_init(CPL_INIT_DEFAULT);

/*
 * Create table
 */

table = cpl_table_new(rows);

/*
 * Compute depth of column of arrays, and create column of images:
 */

depth = 1;
for (i = 0; i < naxis; i++)
    depth *= size[i];

cpl_table_new_column_array(table, "Images", CPL_TYPE_FLOAT, depth);

/*
 * Set the column dimensions: an array of two elements carries the
 * size in x and y of each image
 */

array = cpl_array_new(naxis, CPL_TYPE_INT);

for (i = 0; i < naxis; i++)
    cpl_array_set_int(array, i, size[i]);

cpl_table_set_column_dimensions(table, "Images", array);

cpl_array_delete(array);

/*
 * Now allocate an external image of the appropriate sizes, and fill
 * it with some data:
 */

image = cpl_image_new(size[0], size[1], CPL_TYPE_FLOAT);
cpl_image_fill_noise_uniform(image, -1, 1);

/*
 * Copy the image to the column element 4
 */

array = cpl_array_wrap_float(cpl_image_get_data(image));
cpl_table_set_array(table, "Images", 4, array);
```

```
cpl_array_unwrap(array);
cpl_image_delete(image);

/*
 * At the end the array can (and must) be unwrapped, since it was
 * physically copied to the table. If efficiency reasons make this
 * duplication of data impracticable, the last two calls may be
 * replaced by:
 *
 *   cpl_table_get_data_array(table, "Images")[4] = array;
 *
 * where the created array is directly "plugged" into the appropriate
 * column element. Of course in this case cpl_array_unwrap() should
 * not be called, and cpl_image_delete(image) should not be used,
 * because it would destroy data that belong also to the table.
 * cpl_image_unwrap(image) should be used instead, to destroy the
 * image data wrapper.
 */

/*
 * Here is an example on how the image could be extracted from the
 * corresponding table element: we assume here that the column
 * dimensions are not known.
 */

naxis = cpl_table_get_column_dimensions(table, "Images");
if (naxis == 2) {

    for (i = 0; i < naxis; i++)
        size[i] = cpl_table_get_column_dimension(table, "Images", i);

    array = cpl_table_get_array(table, "Images", 4);
    image = cpl_image_wrap_float(size[0], size[1],
                                 cpl_array_get_data_float(array));

    /*
     * Process image...
     */

    ...

    /*
     * Cleanup when done. Note that the array must not be released.
     */

    cpl_image_unwrap(image);
}

cpl_table_delete(table);

cpl_end();
```

```
    return 0;
}
```

5.2.5 Statistics

The *cpl_stats* object provided in CPL is a container of different statistics that have been computed. They may have been computed on an image, a matrix, a table column or several table columns, or from many other objects.

For the moment, only functions to create this statistics object from an image or an image window are provided.

The *cpl_stats* object must be deallocated with *cpl_stats_delete()*, and can be saved in a text file with *cpl_stats_dump()*.

5.2.6 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 5.2.7).

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 ;

    cpl_init(CPL_INIT_DEFAULT);

    /*
     * Load 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_load("myfile.txt");

    /* Compute the mean of the vector */
    mean = cpl_vector_get_mean(vect);

    /* Subtract the mean */
    cpl_vector_subtract_scalar(vect, mean);

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

```
    /* Delete */
    cpl_vector_delete(vect);

    /* Return */
    cpl_end();
    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.
- 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.

5.2.7 Bivectors

The *cpl_bivector* object is simply composed with two *cpl_vector* objects. Its goal is typically to contain a list of positions in an image, a list of offsets, a list of points defining a one-dimension signal, etc...

The functionality provided by the bivector methods includes:

- A constructor and a destructor.
- Accessor functions to its two vectors.
- Read/write functionalities.
- Interpolation function.

The accessor functions give access to the vectors, so that all the *cpl_vector* methods are available to the bivector members.

5.2.8 Polynomials

A n dimensions polynomial object (*cpl_polynomial*) is provided in CPL, with several methods to create it, deallocate it, set its coefficients, and do some simple operations on it.

5.2.9 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.h>
...
int main()
{
    ...
    cpl_matrix *matrix;
    double     *data_buffer;
    int         number_of_rows    = 20;
    int         number_of_columns = 4;
    double      value;
    ...

    cpl_init(CPL_INIT_DEFAULT);

    ...
    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
     */

    data_buffer = cpl_matrix_get_data(matrix);
    ...
    cpl_matrix_delete(matrix);
    ...

    cpl_end();

    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.
- Inversion.

The methods to support these and other operations are all described in detail in the *CPL Reference Manual* [1], but in the following some of the functionalities are explained with the help of one single example, namely the solution of a redundant linear system, *i.e.*, a system with too many linear equations or too many unknowns. In this example a rather simplistic approach is applied: note that the implementation of the higher-level function `cpl_matrix_solve_normal()` is by far more efficient and sofisticated.

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

$$\mathbf{A}\mathbf{x} = \mathbf{B}$$

is defined, where **x** is the column matrix of the unknowns. The pseudo-inverse solution of this system (in a least-square sense) is given by

$$\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{B}$$

In the following code, a system of 100 equations in 10 unknowns is solved:

```
...
#include <cpl.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;
    ...

    cpl_init(CPL_INIT_DEFAULT);
    ...

    /* 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_create(coeff);
    m1 = cpl_matrix_product_create(t_coeff, coeff);
    m2 = cpl_matrix_invert_create(m1);
    if (m2 == NULL)                         /*  Singular matrix  */
        return 1;
    m3 = cpl_matrix_product_create(t_coeff, nonhomo);
    solution = cpl_matrix_product_create(m2, m3);

    /* Cleanup */

    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);
    ...

    cpl_end();
    return 0;
}
```

5.2.10 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.
- 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()`
- `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
- 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 output the messages to a logfile, a call to cpl_msg_set_log_level() is also required, while output to terminal is automatically enabled at a verbosity level CPL_MSG_INFO; the function cpl_msg_set_level() may be used just to modify this default verbosity. The name of the created log file may be set with the function cpl_msg_set_log_name() before calling cpl_msg_set_log_level(), otherwise it is left to a default ".logfile".

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_set_time_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_set_domain_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_set_component_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 breaking long lines of text to the actual terminal width or to a specific maximum value, and will always add a new line character at the end of any message if it is missing. If the width of the output device cannot be determined, lines of text are not splitted when written to output. If line breaking is not wanted, the function cpl_msg_set_width() should be called specifying a non positive width. To enforce breaking a line of text, new line characters can always be inserted within the message.

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

```
...
#include <cpl.h>
...
int main()
{
```

```
    ...
    char domain[] = "Example";
    char component[] = "messaging";
    ...

    cpl_init(CPL_INIT_DEFAULT);

    /*
     * Initialising the messaging system. Messages are sent both to
     * terminal and to logfile.
     */

    cpl_msg_set_time_on();
    cpl_msg_component_on();
    cpl_msg_set_domain(domain);
    cpl_msg_set_domain_on();
    cpl_msg_set_level(CPL_MSG_WARNING);
    cpl_msg_set_log_level(CPL_MSG_DEBUG);

    /*
     * Printing something...
     */

    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_end();

    return 0;
}
```

A complete description of the functions available in the messaging component is given in the on-line *CPL Reference Manual* [1].

5.2.11 Error handling

This component provides a means to detect, display and recover from errors in CPL-functions. It also allows the CPL API programmer to write functions that sets errors.

A CPL error consists of the following information:

- The CPL error code, an *enum* that defines the type of error, similarly to the *errno* variable of the standard C library. The possible values of CPL error code include `CPL_ERROR_NONE`, which equals zero.
- A human-readable text describing the type of error, optionally followed by more details about the specific error. This text may be used by the caller for error reporting.

- The name of the function in which the error occured.
- The name of the source file in which the error occured.
- The line number where the error occured in that source file.

The CPL errorstate consists of the (possibly empty) sequence of CPL errors that has occurred and from which no recovery has been done.

The most recent CPL error can be queried with these functions:

- `cpl_error_get_code()`.
- `cpl_error_get_message()`.
- `cpl_error_get_function()`.
- `cpl_error_get_file()`.
- `cpl_error_get_line()`.
- `cpl_error_get_where()`, which combines the location information from the above three functions into a single, colon-separated string.

CPL functions modify the CPL error code as follows:

- The CPL error code is initialized by the call to `cpl_init()`. If no error happens in `cpl_init()`, then `cpl_error_get_code()` returns `CPL_ERROR_NONE`. (If an error does happen in `cpl_init()`, then it is unlikely that the application can do anything useful with CPL).
- If no error occurs in other CPL functions, then the CPL errorstate and therefore the return value of `cpl_error_get_code()` is unchanged.
- If an error does happen in a CPL function, a new CPL error is created and appended to the CPL errorstate and the return value of `cpl_error_get_code()` is updated accordingly.
- The behaviour of all CPL functions, except those that implement the CPL error handling, is not affected by the CPL errorstate, i.e. the CPL errorstate is not an input to these functions. This means that if an error has happened, CPL functions can still be called to get information about the conditions that have led to the error.

In general CPL functions do not themselves display any error messages, instead it is left to the caller to decide if and how to display error messages.

If `cpl_error_get_code()` returns `CPL_ERROR_NONE` the CPL errorstate is said to be empty or clean. In this case calls to the other accessors of the CPL error handling are still allowed, but they provide no meaningful information.

Some CPL functions are of type `cpl_error_code`. A function of this type returns `CPL_ERROR_NONE` if it did not create a new CPL error. If it did create one or more new CPL errors, it returns the CPL error code of the most recent error.

Other CPL functions have return values that indicate if a new CPL error has been created, e.g. most of the CPL functions that return a pointer.

A third group of CPL functions cannot indicate with their return value if an error ocurred. If the CPL errorstate is clean prior to a call to such a function, then `cpl_error_get_code()` can indicate if an error was created. This method cannot be used if the CPL errorstate contains errors prior to the call.

In this case the most general method for error detection has to be used. This consists of defining a variable of type `cpl_errorstate` to the value of the errorstate prior to the call, and then comparing this value to the errorstate after the call. Thus to robustly detect whether an error has happened in a call to the function that returns the minimum pixel value in a CPL image, one could do:

```
cpl_errorstate prestate = cpl_errorstate_get();
double valmax = cpl_image_get_max(image);

if (cpl_errorstate_is_equal(prestate)) {
  /* No error happened in cpl_image_get_max(). */
} else {
  /* An error happened in cpl_image_get_max(). */
}
```

In some cases a CPL application can recover from a (sequence of) CPL error(s).

There are two methods for doing this.

The first and simplest consists of a single call to `cpl_error_reset()`, which will empty the entire CPL errorstate and thus cause a subsequent call to `cpl_error_get_code()` to return `CPL_ERROR_NONE`. This method can be used if the CPL errorstate is guaranteed to be clean prior to the code that created the error(s).

The second and more general method consists of defining a variable of type `cpl_errorstate` to the value of the errorstate prior to the code from which recovery is possible, and then setting the errorstate back to this value after the execution of the code from which the recovery is to be done.

For example:

```
cpl_errorstate prestate = cpl_errorstate_get();

my_function();

if (cpl_errorstate_is_equal(prestate)) {

  /* No error happened in my_function() */
  /* - thus no recovery is needed

} else {

  /* Error(s) happened in my_function(). */
```

```
  /* - set the errorstate back to what it was before and discard the
       information about the errors that happened in my_function(). */

  cpl_errorstate_set(prestate);

}

assert( cpl_errorstate_is_equal(prestate) );
```

The CPL errorstate can contain a limited number of CPL errors. This number is defined by the cpp-macro `CPL_ERROR_HISTORY_SIZE` which currently has a default size of 20. The default size of `CPL_ERROR_HISTORY_SIZE` ensures that no CPL function overflows the errorstate.

If more than `CPL_ERROR_HISTORY_SIZE` CPL errors are appended to the CPL errorstate, then the information about the oldest CPL errors is lost. This has implications for error recovery, which are best explained with an example that includes the above code example. Suppose that CPL has been built with the default value (20) for `CPL_ERROR_HISTORY_SIZE`, that the above `prestate` has been defined when the errorstate contains 5 CPL errors, and that the above `my_function()` appends 30 CPL errors to the CPL errorstate.

After the recovery, the CPL errorstate again consists of 5 errors, i.e. the above assertion, `cpl_errorstate_is_equal(prestate)`, still holds. Also, when prestate was defined, `cpl_error_get_code()` would return a value different from `CPL_ERROR_NONE`. At the point of the above `assert()`, `cpl_error_get_code()` would still return a value different from `CPL_ERROR_NONE`.

The information that has been lost at the recovery are:

- `cpl_error_get_code()` returns `CPL_ERROR_UNSPECIFIED` regardless of what it returned when `prestate` was defined.
- The text message of the error has been lost.
- All location information about the error has been lost.

If further recovery is done back to an even older error, the same holds for that error.

The sequence of CPL errors in a non-empty CPL errorstate can be displayed using `cpl_errorstate_dump()`. To display the errors that have occurred after a certain point one could do:

```
cpl_errorstate prestate = cpl_errorstate_get();

my_function();

if (cpl_errorstate_is_equal(prestate)) {
  /* No error happened in my_function() */
} else {
  /* Error(s) happened in my_function(). */
```

```
  /* Dump them all in chronological order, oldest first */
  cpl_errorstate_dump(prestate, CPL_FALSE, cpl_errorstate_dump_one);
}
```

`cpl_errorstate_dump()` takes a boolean, if this evaluates to `CPL_TRUE`, then the order of the dump is reversed. `cpl_errorstate_dump()` takes a function pointer, each CPL error is dumped with a call to that function. To get the default dump, the caller may use `cpl_errorstate_dump_one` or `NULL`. `cpl_errorstate_dump_one` dumps using the CPL messaging system at error level.

The CPL application may define its own functions for dumping a CPL error, the CPL application programmer is referred to the documentation of `cpl_errorstate_dump_one()` for more details about this.

If the dump consists of more than `CPL_ERROR_HISTORY_SIZE` errors, then all but the newest `CPL_ERROR_HISTORY_SIZE` will be displayed with the error code `CPL_ERROR_UNSPECIFIED` and empty text and location information.

The currently available CPL error codes are:

`CPL_ERROR_NONE` No error

`CPL_ERROR_UNSPECIFIED` An unspecified error

`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 permission denied

`CPL_ERROR_BAD_FILE_FORMAT` Bad file format

`CPL_ERROR_FILE_ALREADY_OPEN` File already open

`CPL_ERROR_FILE_NOT_CREATED` File cannot be created

`CPL_ERROR_FILE_NOT_FOUND` File not found

`CPL_ERROR_DATA_NOT_FOUND` Data not found

`CPL_ERROR_ACCESS_OUT_OF_RANGE` Access beyond boundaries

`CPL_ERROR_NULL_INPUT` Null input data

`CPL_ERROR_INCOMPATIBLE_INPUT` Input data do not match

`CPL_ERROR_ILLEGAL_INPUT` Illegal input

`CPL_ERROR_ILLEGAL_OUTPUT` Illegal output

`CPL_ERROR_UNSUPPORTED_MODE` Unsupported mode

`CPL_ERROR_SINGULAR_MATRIX` Singular matrix

`CPL_ERROR_DIVISION_BY_ZERO` Division by zero

`CPL_ERROR_TYPE_MISMATCH` Type mismatch

`CPL_ERROR_INVALID_TYPE` Invalid type

`CPL_ERROR_CONTINUE` The iterative process did not converge

`CPL_ERROR_EOL` A user-defined error

`CPL_ERROR_EOL` is guaranteed to not be used within CPL itself, and to be greater than any of the CPL other error codes. `CPL_ERROR_EOL` can therefore be used by the CPL application to extend the error handling with new error codes.

Here is an example of a program with CPL error handling.

```
#include <cpl.h>

cpl_error_code my_func(void);

int main(void)
{
    cpl_errorstate prestate;

    cpl_init(CPL_INIT_DEFAULT);

    prestate = cpl_errorstate_get();

    if (my_func() != CPL_ERROR_NONE) {
        /* At this point error recovery is not possible
           - instead dump the error state. */

        cpl_msg_error(cpl_func, "my_func() failed:");
        cpl_errorstate_dump(prestate, CPL_FALSE, cpl_errorstate_dump_one);
    }

    return cpl_error_get_code() ? EXIT_FAILURE : EXIT_SUCCESS;
}

cpl_error_code my_func(void)
{

    /* Declarations needed for error handling */
    cpl_errorstate  prestate = cpl_errorstate_get();
    cpl_error_code  status;

    /* Other declarations */
    cpl_matrix     *matrix = cpl_matrix_new(10, 10);
    cpl_matrix     *inverse;
    double          mean;
```

```
/*
 * Propagate the error from a function of type cpl_error_code.
 */

status = my_matrix_fill(matrix);
if (status != CPL_ERROR_NONE) {
    /* Free memory and propagate the unrecoverable error */
    cpl_matrix_delete(matrix);
    return cpl_error_set_message(cpl_func, cpl_error_get_code(),
                                 "Could not fill matrix");
}

/*
 * Propagate the error in a function returning a valid pointer
 * on success, or a NULL in case of failure.
 */

inverse = cpl_matrix_invert_create(matrix);
if (inverse == NULL) {
    /* Free memory and propagate the unrecoverable error */
    cpl_matrix_delete(matrix);
    return cpl_error_set_message(cpl_func, cpl_error_get_code(),
                                 "Could not invert matrix");
}

/*
 * Propagate error in a function whose return value cannot
 * indicate the error status.
 */

mean = cpl_matrix_get_mean(matrix);
if (!cpl_errorstate_is_equal(prestate)) {
    /* Free memory and propagate the unrecoverable error */
    cpl_matrix_delete(matrix);
    cpl_matrix_delete(inverse);
    return cpl_error_set_message(cpl_func, cpl_error_get_code(),
                                 "Could not compute mean of matrix");
}

/*
 * Handle failure of a function of type cpl_error_code.
 * A switch may be used to catch specific error codes, which
 * can be handled. In this example, the errors
 * CPL_ERROR_DIVISION_BY_ZERO and CPL_ERROR_CONTINUE can be handled,
 * while others cannot. Note that, for those errors that can be
 * handled the errors are discarded from the CPL error state.
 */

status = my_matrix_correction(matrix, inverse, mean);
```

```
    switch (status)
        {
        case CPL_ERROR_NONE:
            break; /* No action needed */

        case CPL_ERROR_DIVISION_BY_ZERO:
            cpl_msg_warning(cpl_func, "Correction caused division by zero, "
                            "matrix correction skipped.");
            cpl_errorstate_set(prestate); /* Recover from error(s) */
            break;

        case CPL_ERROR_CONTINUE:
            cpl_msg_warning(cpl_func, "Correction did not converge, "
                            "trying robust method.");

            cpl_errorstate_set(prestate); /* Recover from error(s) */

            my_matrix_correction_robust(matrix, mean);
            assert( cpl_errorstate_is_equal(prestate) );

            break;

        default:
            /* Free memory and propagate the unrecoverable error */
            cpl_matrix_delete(matrix);
            cpl_matrix_delete(inverse);

            return cpl_error_set_message(cpl_func, cpl_error_get_code(),
                                         "Correction caused an unexpected error");
        }

    /* Free memory and return successfully */
    cpl_matrix_delete(matrix);
    cpl_matrix_delete(inverse);

    return CPL_ERROR_NONE;

}
```

The functions to support error handling are all described in detail in the online *CPL Reference Manual* [1].

5.2.12 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 5.2.13). Note the difference between a *cpl_property* (an atomic variable storage mechanism) and a *cpl_propertylist* (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.

5.2.13 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_propertylist* make use of the *cpl_property* type (see Section 5.2.12), 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_propertylist* 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. *Properties* which belong to a property list can be extracted using the functions `cpl_propertylist_get_property()` and its constant related version, `cpl_propertylist_get_property_const()`.

```
#include <cpl.h>

...

int main()
{

  ...

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


  cpl_propertylist *list;

  ...

  cpl_init(CPL_INIT_DEFAULT);

  ...
```

```
  list = cpl_propertylist_new();

  ...

  cpl_propertylist_append_int(list, "MyInt", 42);
  cpl_propertylist_append_float(list, "MyFloat", 1.e-6);
  cpl_propertylist_append_string(list, "MyString", "text");

  ...

  i = cpl_propertylist_get_int(list, "MyInt");
  f = cpl_propertylist_get_float(list, "MyFloat");
  s = cpl_propertylist_get_string(list, "MyString");

  ...

  cpl_propertylist_delete(list)

  ...

  cpl_end();

  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.

5.2.14 Plotting

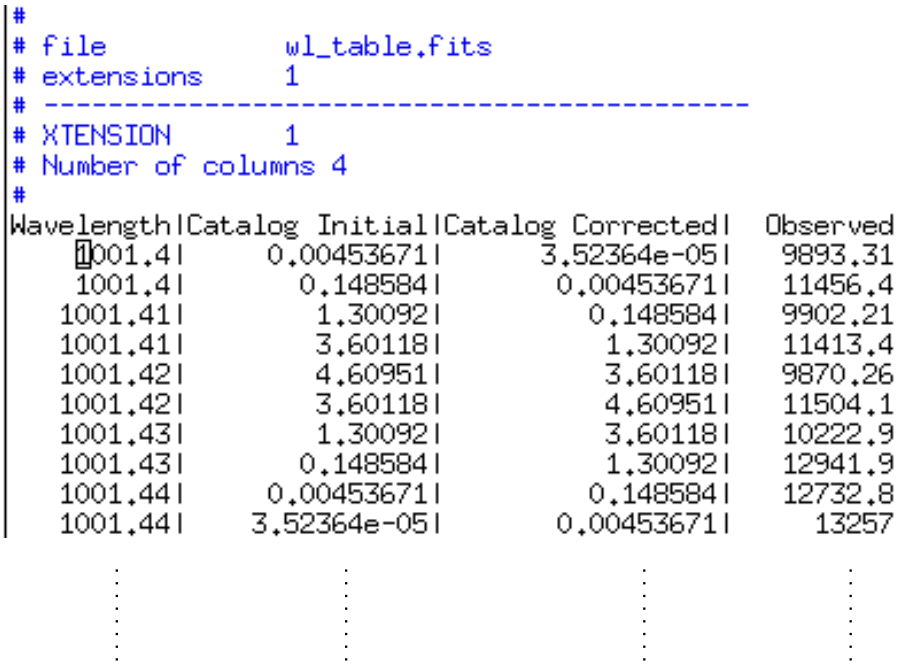
For a number of CPL objects, we provide simple plotting functionalities by using *gnuplot* internally. In order for these functionalities to work properly, the only requirement is to have *gnuplot* installed on your system. If it is not, the function will not set any error, but will just remain without any effect.

As an example, the following code shows how to overplot several columns of the table (see Figure 11) produced by the CPLDRS *cpl_wlcalib_xc_best_poly()* function.

```
int example_plot_spc_table(const cpl_table * spc_table)
{
    cpl_vector      **  vectors ;

    /* Test entries */
    if (spc_table == NULL) return -1 ;

    /* Initialise */
    nsamples = cpl_table_get_nrow(spc_table) ;
```



```
#
# file           wl_table.fits
# extensions     1
# ---------------------------------------------
# XTENSION       1
# Number of columns 4
#
Wavelength|Catalog Initial|Catalog Corrected|  Observed
    1001.4|     0.00453671|      3.52364e-05|   9893.31
    1001.4|       0.148584|       0.00453671|   11456.4
   1001.41|         1.30092|         0.148584|   9902.21
   1001.41|         3.60118|          1.30092|   11413.4
   1001.42|         4.60951|          3.60118|   9870.26
   1001.42|         3.60118|          4.60951|   11504.1
   1001.43|         1.30092|          3.60118|   10222.9
   1001.43|       0.148584|          1.30092|   12941.9
   1001.44|     0.00453671|         0.148584|   12732.8
   1001.44|    3.52364e-05|       0.00453671|     13257
```

Figure 11: Table to plot

```
vectors = cpl_malloc(4*sizeof(cpl_vector*)) ;
vectors[0] = cpl_vector_wrap(nsamples,
        cpl_table_get_data_double((cpl_table*)spc_table,
            "Wavelength"));
vectors[1] = cpl_vector_wrap(nsamples,
        cpl_table_get_data_double((cpl_table*)spc_table,
            "Catalog Initial"));
vectors[2] = cpl_vector_wrap(nsamples,
        cpl_table_get_data_double((cpl_table*)spc_table,
            "Catalog Corrected"));
vectors[3] = cpl_vector_wrap(nsamples,
        cpl_table_get_data_double((cpl_table*)spc_table,
            "Observed")) ;

irplib_vectors_plot("set grid;set xlabel 'Wavelength (nm)';",
    "XC 1-Initial cat/2-Corrected cat/3-Observed' w lines",
    "", (const cpl_vector **)vectors, 4);

cpl_vector_unwrap(vectors[0]) ;
cpl_vector_unwrap(vectors[1]) ;
cpl_vector_unwrap(vectors[2]) ;
cpl_vector_unwrap(vectors[3]) ;
cpl_free(vectors) ;
return 0 ;
```

```
}
```

The figure 12 shows how appears the plot generated by the example function when used on the example table.

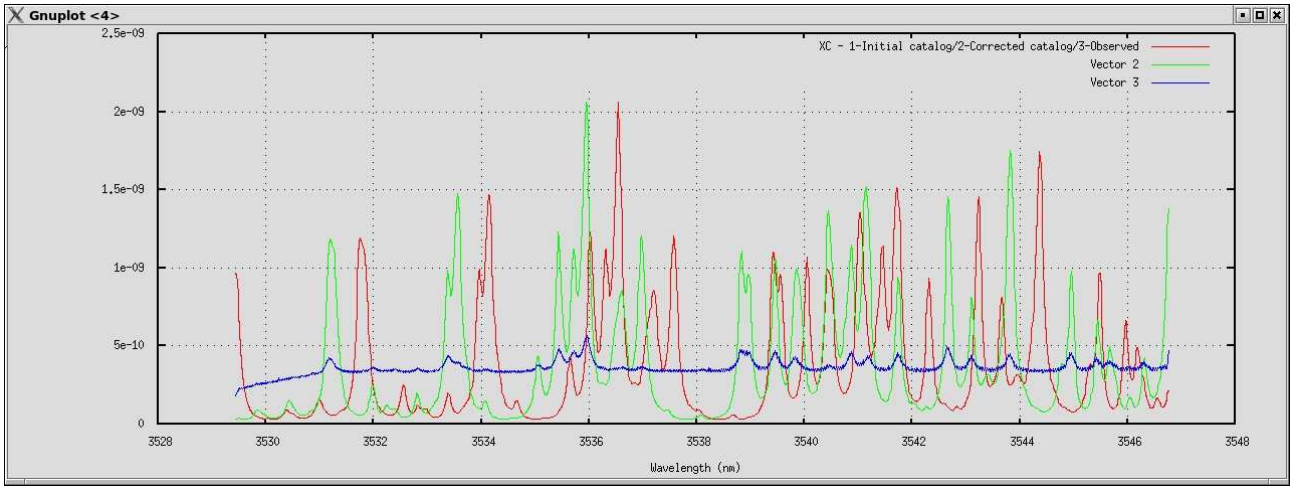


Figure 12: Effect of the plotting function with the table

5.3 The CPL interfaces in *libcplui*

5.3.1 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.h>

...

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

    cpl_frame *product_frame;
```

```
    cpl_image_add(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;

}
```

5.3.2 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.h>

...

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

    ...

    cpl_frame *bias_frame,
    cpl_frame *result_frame;
    cpl_image *bias;

    ...

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

    result_frame = cpl_frame_new();

    ...

    cpl_frameset_insert(set, result_frame);

    ...
```

```
    return set;

}
```

5.3.3 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 a 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_parameterlist*, 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.

It is possible to extract/insert parameters within parameter lists. For a complete documentation of the parameter component please refer to the online *CPL Reference Manual* [1].

```
#include <cpl.h>

...

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

    cpl_parameterlist *plist = cpl_parameterlist_new();
    cpl_parameter *p;


    p = cpl_parameter_new_value("config.integer_value",
                                CPL_TYPE_INT,
```

```
                                "An integer value",
                                "config",
                                0);

        cpl_parameter_set_int(p, i);
        cpl_parameterlist_append(plist, p);

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

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

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

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

        return plist;

    }
```

5.4 Standard data reduction algorithms in *libcpldrs*

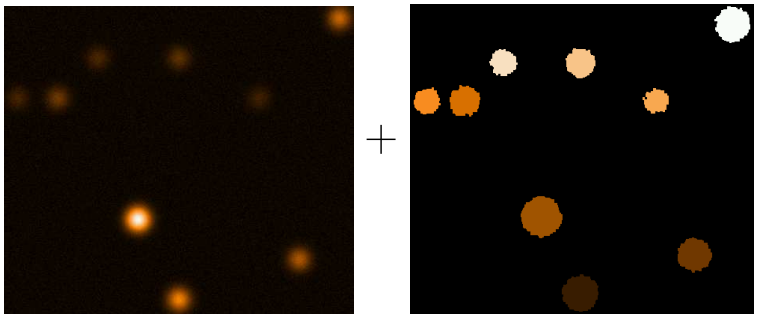
The CPL *libcpldrs* library provides standard astronomical data reduction algorithms.

5.4.1 Apertures

The *cpl_apert* object can contain information or statistics of a list of objects or zones in an image. The function that creates this object is *cpl_apertures_new_from_image()*. It takes as input the image in which the objects are, and a labels image (an integer image) that defines the different zones or objects positions in the input image. This labels image has the same size as the input image and identifies with its labels the different zones, negative values identify the background.

So if the labels image contains pixels with n different positive values, *cpl_apertures_new_from_image()* will create a *cpl_apert* object containing n different apertures with various statistics computed on each of them (see Figure 13).

The objects detection itself is done by the computation of the labels image, this here is just statistics computation of the already specified detected objects.



cpl_apertures_new_from_image()

#	X	Y	XCENTROID	YCENTROID	pix	max	min	mean	med	dev	flu
1	255.9	102.1	255.9	102.0	993	8.76	1.21	4.01	3.65	1.88	3980.71
2	409.1	152.9	409.1	153.0	733	6.26	1.25	3.22	3.08	1.15	2357.11
3	203.8	203.9	204.0	204.0	1408	16.69	1.22	6.33	5.19	3.98	8914.26
4	102.3	358.0	102.2	357.9	365	3.99	1.23	2.46	2.47	0.65	896.80
5	52.3	358.6	52.3	358.6	33	2.62	1.27	2.03	2.01	0.44	66.88
6	356.5	358.5	356.5	358.4	16	2.38	1.26	1.82	1.77	0.37	29.13
7	255.8	409.3	255.8	409.3	258	3.61	1.22	2.17	2.14	0.59	560.59
8	153.3	405.1	153.2	405.1	45	2.81	1.22	2.14	2.21	0.41	96.42
9	459.9	460.2	459.9	460.1	828	6.95	1.22	3.47	3.27	1.39	2875.25

Figure 13: Usage of *cpl_apertures_new_from_image()*

However, this module provides a very simple objects detection function named *cpl_apertures_extract()*. You just need to pass a list of sigma values (in a *cpl_vector()*), and the function will apply a sigma threshold to find objects in the passed image. It will internally create the labels image, call the *cpl_apertures_new_from_image()* function and return the *cpl_apert* object. If nothing is detected with the first sigma value, the second is used and so on until something is detected. *cpl_apertures_extract_sigma()* does it with only one passed sigma, and *cpl_apertures_extract_window()* does it on a window of the image.

Besides, this module provides functions to sort the different apertures according to the number of pixels, the maximum value or the flux.

5.4.2 Detectors

This part contains high-level functions commonly used to get detector characteristics like the non-linearity or the read-out noise, or to correct detector defaults like the bad pixels.

1. Read-out noise computation

   The noise computed by the functions *cpl_flux_get_noise_window()* and *cpl_flux_get_noise_ring()* is the median of the standard deviation values computed in a number of small windows scattered optimally using a Poisson law in the specified region of the input image (a window or a ring).

2. Bad pixels reconstruction

   The *cpl_detector_interpolate_rejected()* recomputes the bad pixels of an image by using the good pixels in the neighborhood. An iterative procees is used until all bad pixels have been corrected.

5.4.3 Geometrical transformations

The functions currently contained in this part can combine an image list into a single image. The input image list is typically a jitter observation (observation technique commonly used in infra red to remove the strong background) and the fuction shifts and adds the images together.

The function *cpl_geom_img_offset_combine()* is very flexible, the offsets can be specified or not, they can be refined or not with cross-correlation, the anchor point used for the cross-correlation can be specified or not, sigma values can be specified if the function needs to find itself this anchor point, and the stacked image can be the union or the intersection of the input images. The diagram in Figure 14 shows what the function does.

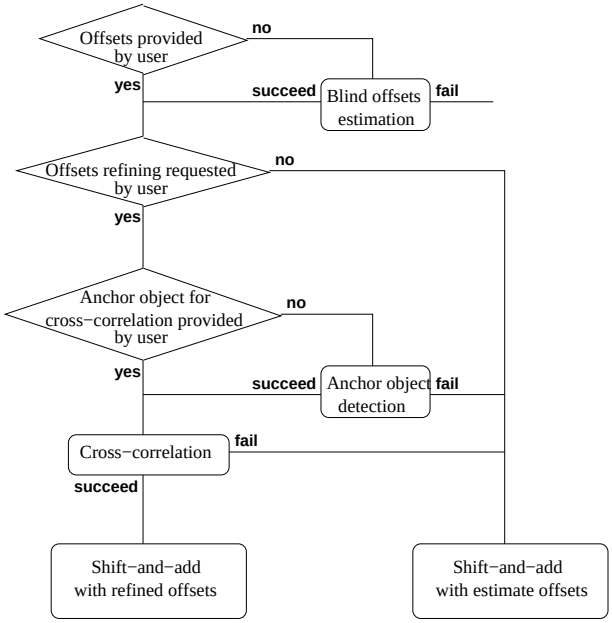


Figure 14: *cpl_geom_img_offset_combine()* behaviour

5.4.4 Photometry

This part currently contains a function (*cpl_photom_fill_blackbody()*) that computes the Planck black-body radiance.

5.4.5 Nonlinear fitting

This part contains one high-level function for general nonlinear fitting.

1. Levenberg-Marquardt

   The function *cpl_fit_lvmq()* provides a LeVenberg-MarQuardt routine for fitting nonlinear one-dimensional or multi-dimensional data.

5.4.6 World Coordinate System

The World Coordinate System facility provided by CPL offers a way to create and manipulate the WCS descriptions for a given image. At the heart of *cpl_wcs* is Marc Calabretta's *WCSLIB* package available from (`http://www.atnf.csiro.au/people/mcalabre/WCS/`). The current implementation of *cpl_wcs* allows the user to

- load a WCS from a propertylist containing a valid FITS WCS description,
- do basic coordinate conversions
- use standard object positions to define an image WCS.

A typical use for *cpl_wcs* would be to work out the RA and Dec of an object given its physical coordinates on an image. In the following fragment the Cartesian coordinates of two objects is given in the static double array `phys`. The header of the original image is parsed into a propertylist and the WCS information is recovered from it. The physical coordinates are wrapped in a *cpl_matrix* structure and passed to the conversion routine. Output is another *cpl_matrix* structure with the world coordinates of the two objects. The FITS header of the input image will determine the type of coordinates produced and the projection geometry used. Thus this conversion routine could be used to produce any type of world coordinate that is supported by FITS.

```
#include <cpl.h>

...

static double phys[] = {382.252,  36.261,
                        18.097,  738.428};
int main()
{

  ...

  char *filename;
  const cpl_wcs *wcs;
  const cpl_propertylist *plist;
  cpl_matrix *from,*to;
  cpl_array *status;

  ...

  cpl_init(CPL_INIT_DEFAULT);

  ...

  plist = cpl_propertylist_load(filename,1);
  wcs = cpl_wcs_new_from_propertylist(plist);
  from = cpl_matrix_wrap(2,2,phys);
  cpl_wcs_convert(wcs,from,&to,&status,CPL_WCS_PHYS2WORLD);
```

```
  ...


  cpl_matrix_unwrap(from);
  cpl_matrix_delete(to);
  cpl_array_delete(status);
  cpl_propertylist_delete(plist);
  cpl_wcs_delete(wcs);

  ...

  cpl_end();

  return 0;

}
```

The *cpl_wcs_convert* routine can do conversions between three types of coordinates:

physical A physical location of an object in pixel space.

world Space/time coordinates of an object in a given astronomical system.

standard An intermediate coordinate defined as an offset from the defined world coordinate system reference point. This will be in the natural coordinate units for the WCS.

and currently supports several conversion modes:

CPL_WCS_PHYS2WORLD Physical coordinates are converted to world coordinates. The output coordinate system depends entirely on the values defined in the *cpl_wcs* structure and ultimately from the FITS header from which it was derived.

CPL_WCS_WORLD2PHYS World coordinates are converted to physical coordinates. It is entirely up to the user to ensure that the coordinates given are consistent with the WCS coordinate geometry that is provided by the input FITS header.

CPL_WCS_WORLD2STD World coordinates are converted to standard coordinates.

CPL_WCS_PHYS2STD Physical coordinates are converted to standard coordinates.

The WCS facility also offers a routine to fit a two-dimensional WCS to a list of objects with known world and physical coordinates *cpl_wcs_platesol*. The desired form of the WCS is defined by an input propertylist. In most cases this would probably be parsed from the header of an input FITS image, but in fact could also be built from scratch by the user. A full explanation of the elements needed to define a WCS in FITS is way beyond the scope of this manual and the reader is referred to the web pages of the FITS support office at NASA/GSFC (`http://fits.gsfc.nasa.gov/fits_wcs.html`) and to the references therein. The output propertylist contains the new FITS WCS description. It is worth noting that this routine will fit for offset,

scale and rotation, but will not fit any of the parameters for the projection geometry. These must be fixed in the input WCS description.

Accessor functions are not included in the *cpl_wcs* API. Any modifications that the user wishes to make to a WCS must be done to the input propertylist before it is parsed into the *cpl_wcs* structure.

5.5 ESO/DFS specific routines in *libcpldfs*

The functions contained in this library implement DFS specific requirements on keywords for pipeline products. These functions are called by all pipelines, and insure these pipeline to have products that are compliant with the last requirements.

[1] Common Pipeline Library Reference manual.

[2] P. Ballester. Data Flow for VLT/VLTI Instruments – Delivrables Specification. 2004.

[3] P. Grosbol P. Ballester, K. Banse. Data Flow Pipeline and Quality Control - Users Manual. 1999.

[4] Eso DICB – Data Interface Control Document. 1996.

[5] Recommended C Style and Coding Standards.

A The PDRM source code

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

```
#include <cpl.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
 */

static int myplugin_create(cpl_plugin *);
static int myplugin_exec(cpl_plugin *);
static 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,
                    CPL_PLUGIN_TYPE_RECIPE,
                    "myplugin",
                    "Do basic arithmetics on two images",
                    myplugin_help,
                    "Gill Bates",
                    "gbates@macrohard.com",
```

```
                    "GPL",
                    myplugin_create,
                    myplugin_exec,
                    myplugin_destroy);

    cpl_pluginlist_append(list, plugin);

    return 0;

}


static int
myplugin_create(cpl_plugin *plugin)
{

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


    recipe->parameters = cpl_parameterlist_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, CPL_PARAMETER_MODE_CLI, "op");
    cpl_parameterlist_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_parameterlist_delete(recipe->parameters);

    return 0;
}
```

B 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: cpl_image_io.h,v 1.48 2005/02/16 17:56:33 yjung Exp $
 *
 * This file is part of the ESO Common Pipeline Library
 * Copyright (C) 2001-2004 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: yjung $
 * $Date: 2005/02/16 17:56:33 $
 * $Revision: 1.48 $
 * $Name:  $
 */

#ifndef CPL_IMAGE_IO_H
#define CPL_IMAGE_IO_H

/*-----------------------------------------------------------------------------
                                   New types
 -----------------------------------------------------------------------------*/

typedef struct _cpl_image_ cpl_image;

/*-----------------------------------------------------------------------------
```

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

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <limits.h>

#include "cpl_io.h"
#include "cpl_propertylist.h"
#include "cpl_mask.h"

CPL_BEGIN_DECLS

/*----------------------------------------------------------------------------
                                    Define
 ---------------------------------------------------------------------------*/

#define CPL_PIXEL_MAXVAL       (double)(LONG_MAX)
#define CPL_PIXEL_MINVAL       (double)(LONG_MIN)

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

/* Image constructors */
cpl_image * cpl_image_new(int, int, cpl_type);
cpl_image * cpl_image_wrap_double(int, int, const double *) ;
cpl_image * cpl_image_wrap_float(int, int, const float *) ;
cpl_image * cpl_image_wrap_int(int, int, const int *) ;
cpl_image * cpl_image_load(const char *, const cpl_type, const int, const int) ;
cpl_image * cpl_image_new_from_mask(const cpl_mask *) ;

...


CPL_END_DECLS

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

C Naming conventions

The naming conventions are described in section 4.7.

Qualifiers The following words are permitted as qualifiers in `get`/`set` operations:

- `absflux`
- `alias`
- `api`
- `author`
- `bool`
- `bottom`
- `centroid`
- `char`
- `class`
- `code`
- `coeff`
- `column`
- `comment`
- `component`
- `context`
- `copyright`
- `cputime`
- `data`
- `default`
- `degree`
- `deinit`
- `description`
- `determinant`
- `dimension`
- `domain`
- `double`
- `email`
- `enum`
- `exec`
- `file`
- `filename`

- `first`
- `float`
- `flux`
- `format`
- `frame`
- `function`
- `fwhm`
- `group`
- `help`
- `id`
- `indentation`
- `info`
- `init`
- `int`
- `interpolated`
- `invalid`
- `keyword`
- `last`
- `left`
- `level`
- `line`
- `log`
- `long`
- `macro`
- `max`
- `maxpos`
- `mean`
- `median`
- `message`
- `min`
- `minpos`
- `name`
- `ncol`
- `next`
- `nextensions`
- `noise`

- `npix`
- `nrow`
- `range`
- `right`
- `size`
- `sizeof`
- `sqflux`
- `stdev`
- `string`
- `synopsis`
- `tag`
- `time`
- `top`
- `type`
- `unit`
- `version`
- `where`
- `width`
- `x`
- `y`

The following words are permitted as qualifiers for other operations:

- `1d`
- `2d`
- `after`
- `all`
- `array`
- `blackbody`
- `bool`
- `but`
- `by`
- `char`
- `coarse`
- `column`
- `columns`
- `combine`

- `context`
- `create`
- `data`
- `diagonal`
- `double`
- `echelon`
- `empty`
- `enabled`
- `enum`
- `fine`
- `fits`
- `float`
- `forward`
- `frame`
- `frameset`
- `from`
- `gaussian`
- `identity`
- `image`
- `int`
- `invalid`
- `kernel`
- `less`
- `linear`
- `log`
- `long`
- `lowpass`
- `mask`
- `median`
- `more`
- `morpho`
- `noise`
- `normal`
- `overwritable`
- `polynomial`
- `power`

- `product`
- `property`
- `range`
- `regexp`
- `rejected`
- `row`
- `rowcolumn`
- `rows`
- `saa`
- `scalar`
- `segment`
- `selected`
- `sigma`
- `small`
- `stdev`
- `string`
- `structure`
- `subsample`
- `tag`
- `tags`
- `test`
- `to`
- `type`
- `valid`
- `value`
- `vectors`
- `window`
- `zero`

Items The following words are permitted as items:

- `accepted`
- `bool`
- `char`
- `create`
- `data`
- `dev`

- `double`
- `flag`
- `float`
- `flux`
- `format`
- `header`
- `image`
- `int`
- `invalid`
- `level`
- `long`
- `macro`
- `mask`
- `max`
- `maxpos`
- `mean`
- `median`
- `min`
- `minpos`
- `name`
- `npix`
- `of`
- `off`
- `on`
- `profile`
- `regexp`
- `rejected`
- `ring`
- `rows`
- `size`
- `stdev`
- `string`
- `strings`
- `type`
- `uniform`
- `unit`
- `window`
- `x`
- `y`

— End of document —