

EURO 3D

Promoting 3D spectroscopy in Europe



Research & Training Network
Sponsored by the European Commission

www.aip.de/Euro3D

Euro3D-LCL libraries

Developer's Guide

Version:	1.0e
Release:	October 18, 2004
Author(s):	Y. COPIN, P. FERRUIT, A. PÉCONTAL-ROUSSET
Contact:	y.copin@ipnl.in2p3.fr

Contents

Revision history	ii
Reference documents	ii
Typographic conventions	ii
1 Introduction	1
1.1 Purpose and scope	1
1.2 Requirements	1
2 Creating & maintaining your own project	2
2.1 Presentation	2
2.2 Setting up the development tree	2
2.3 Developing your project	4
2.3.1 Some comments about the Makefiles	4
2.3.2 Local definitions	5
2.3.3 Adding library sources	5
2.3.4 Adding program sources	6
2.3.5 Using include files	7
2.3.6 Adding a data directory	7
2.4 Using your project	7
2.4.1 Pre-compilation	7
2.4.2 Compiling your project	8
2.4.3 Distributing your project	8
3 Source documentation using doxygen	10
3.1 Standard documentation	10
3.1.1 File documentation	10
3.1.2 Function documentation	10
3.1.3 Other documentation	13
3.1.4 Emacs configuration	13
3.2 doxygen configuration	13
4 Writing the <i>User's Guide</i>	14
A TODOs	15
A.1 Template	15
A.2 Dev's Guide	15
A.3 Misc.	15

List of Figures

1	Layout of the E3D_Template development tree	3
2	doxygen file header	11
3	doxygen function header	12

Revision history

Version	Date	Sections affected	Comments
0.0	10/03/2003	PF	Creation from YC <i>IFU Developer's Guide</i>
1.0a	15/03/2003	all	Internal release for comments inside the LYO team
1.0b	14/04/2003	all	Release of an α version of the document together with the Euro3D-LCL libraries v1.0 α
1.0c	20/04/2003	all	iteration
1.0d	27/09/2004	all	updated template architecture
1.0e	17/10/2004	all	updated template architecture

Reference documents

- [1] *Euro3D-LCL I/O library – Installation guide*, A. Pécontal-Rousset, P. Ferruit and Y. Copin, v1.0
- [2] *Euro3D-LCL I/O library – Cookbooks*, P. Ferruit, A. Pécontal-Rousset and Y. Copin, v1.0
- [3] *Doxygen manual*, D. van Heesch, v1.2.14 (<http://www.stack.nl/~dimitri/doxygen/>)
- [4] *Autoconf & Automake manuals*, Free Software Foundation (<http://www.gnu.org/>)
- [5] *The E3D L^AT_EX package*, Y. Copin, v.1.1

Typographic conventions

- *filename*
- directory/ and package name
- command and shell_variables
- *file_exert*

1 Introduction

1.1 Purpose and scope

This document describes generic rules to develop your own project within the common framework of the Euro3D-LCL software development environment. Section 2 is dedicated to the setup of a valid development tree using the E3D_Template project. Section 3 describes the use of doxygen to document your sources and include any low-level documentations (algorithms, formats, etc.). Section 4 invites you to write the *User's Guide* to your newly created project.

This document is *not*:

- the installation guide of the Euro3D-LCL I/O-library: see [1],
- the cookbook of the Euro3D-LCL I/O-library: see [2],
- a doxygen manual: see [3].
- a GNU-autotools (autoconf, automake, etc.) tutorial: see [4].

1.2 Requirements

It is assumed the reader is coding in C (potentially in C++), and is reasonably familiar with UNIX tools, such as classical commands (`mv`, `tar`, etc.), shell and environment variables, `make` and *makefile* basics.

All the programs used in this tutorial are pretty standard, and should be available in every decent LINUX distribution. However, the GNU-autotools (autoconf, automake, etc. on which is based the Euro3D-LCL framework) experienced a quick and rather brutal evolution phase these last monthes, and it is preferable to start a new development tree using the latest versions (see [4]). E3D_Template-1.0e project is based upon automake-1.8.2 and autoconf-2.59.

Regarding the Euro3D projects and documents, the stable versions are available from the official Euro3D website (<http://www.aip.de/Euro3D>). You may also find development (unstable) versions on the Euro3D-LCL webpage (www-obs.univ-lyon1.fr/lcl).

Prior to setting up your new project, you will need to install the Euro3D-LCL I/O-library. The installation procedure is detailed in [1]. Furthermore, the present tutorial is based upon the E3D_Template project, and you will need the appropriate tarball *E3D_Template-1.0.tar.gz*.

2 Creating & maintaining your own project

In this section, we describe how to create, develop and maintain your own project under the Euro3D-LCL software development environment. Using this architecture allows you to easily and quickly obtain a portable and easy-to-use software bundle.

2.1 Presentation

A *project* is a set of programs and/or libraries gathered in a single structure. Besides some technical directories and files, it consists of one or more *packages* (e.g., for an instrumental project, the packages could be preprocessing, extraction, calibration, etc.), each package providing its own set of programs.

If a given package provides some executables, it is subdivided in three subdirectories (besides technical ones):

- `src/` contains the *main* files of the programs and their associated files,
- `lib/` contains the package-specific library files,
- `incl/` contains the package-specific include files.

The extra technical `obj/` directory gathers the object files of the package (see Fig. 1 for an example). If your package is providing a library, `src/` is of course not required.

Furthermore, if needed, the project can also consist of project-wide libraries and include files. They should then be stored in the `lib/` and `incl/` directories at the level of the other packages.

2.2 Setting up the development tree

We describe now a step-by-step procedure to set up a valid development environment for your own project, labelled hereafter MyProject version 0.0 (see below for a script implementing this procedure). We will use the template architecture provided by the `E3D_Template` project (Fig. 1), as obtained from `E3D_Template-1.0.tar.gz`.

1. Extract the archive file `E3D_Template-1.0.tar.gz` under your `$E3D_PATH`, i.e., at the same level as the Euro3D-LCL I/O-library. This creates a dummy project architecture under the `E3D_Template-1.0/` directory.
2. Rename the `E3D_Template-1.0/` directory to `MyProject-0.0/` (or any other name that suits you). This directory is the *root directory* of your project.
3. For each software package you want to develop as part of your project, you need to create a subdirectory under `pkg/`, e.g., `MyFirstPackage/` and `MySecondPackage/`. To do that, you can rename or copy the `pkg/example/` directory: the structure of the package is then the one presented in 2.1: `lib/`, `incl/` and `src/`. Note that for more generic libraries or include files (i.e., attached to your whole project rather than to a specific package), you should directly use the `pkg/lib/` and `pkg/incl/` directories.
4. Once you do not need it anymore, remove the initial directories `pkg/example/` and `pkg/wizard/`.
5. Edit `pkg/Makefile.am`: initially the first line of this file contains only `SUBDIRS = example wizard`, that is the list of subdirectories to process. As you have changed the directory structure, you must update this file accordingly: on this line, replace `example` by the space-separated list of your packages, e.g.,

```
| SUBDIRS = MyFirstPackage MySecondPackage
```

6. You also need to edit the `configure.ac` file in the root directory of your project:

Architecture of the E3D_Template development tree

+ E3D_Template-1.0/	
- <i>configure.ac</i>	Project configuration file (to be edited)
- <i>Makefile.am</i>	Root <i>Makefile.am</i> (fixed)
- <i>Makefile.in</i>	Automatically generated (<i>overwritten</i>)
+ <i>add_defs/</i>	Contains the <i>makefile</i> definition files
- <i>makedefs</i>	Standard make-definition file (fixed)
- <i>makedefs.local</i>	Project make-definition file (to be edited)
- <i>config.h</i>	Project-wide include file (<i>overwritten</i>)
+ <i>pkg/</i>	Contains the software packages
- <i>Makefile.am</i>	Lists the package directories (to be edited)
- <i>Makefile.in</i>	Automatically generated (<i>overwritten</i>)
+ <i>incl/</i>	Contains the project-wide include files
- <i>template.h</i>	Project-wide include file (to be edited)
+ <i>example/</i>	An example software package directory
- <i>Makefile.am</i>	Lists the source and library directories (may be edited)
- <i>Makefile.in</i>	Automatically generated (<i>overwritten</i>)
+ <i>incl/</i>	Contains the package-wide include files
- <i>example.h</i>	Package-wide include file (to be edited)
+ <i>lib/</i>	Contains the library source files
- <i>makefile.in</i>	Compilation procedures (to be edited)
- <i>makefile</i>	Automatically generated (<i>overwritten</i>)
- <i>libexample.c</i>	Library source file
+ <i>src/</i>	Contains the source files
- <i>makefile.in</i>	Compilation procedures (to be edited)
- <i>makefile</i>	Automatically generated (<i>overwritten</i>)
- <i>E3D_example.c</i>	Main source file
+ <i>obj/</i>	Objects and libraries directory
+ <i>user/</i>	Local installation repository
+ <i>bin/</i>	Executables directory

Figure 1: Layout of the E3D_Template template development tree. The nature of the files (to be edited, fixed or *overwritten*) is indicated in parentheses.

- The line starting with `AC_INIT` is used to specify the full name and the version number of your project (*Euro3D Template* and `1.0` in the *Template* project), as well as an optional tarball name (generated automatically otherwise). Change them to be consistent with the name of the root directory of your project tree, e.g., *MyProject* and `0.0`.
- The line starting with `AC_CONFIG_SRCDIR` contains the relative path from the root of your project to one target, i.e. a single file to be compiled (`pkg/incl/template.h` in the *Template* project). Change it to any valid target of your project.
- The line starting with `AC_CONFIG_FILES` contains the *exhaustive* list of the makefiles to be generated by the `configure` application. In the *E3D_Template* project, besides the technical makefiles (*Makefile* and `pkg/Makefile`) and the documentation makefiles, three makefiles corresponding to the *Template* package are listed:
 - `pkg/example/Makefile`,
 - `pkg/example/lib/makefile`,
 - `pkg/example/src/makefile`.

You need to remove these and put those corresponding to your own software packages. Do not forget any (up to 3 per packages) and be careful about the capital letter: the (technical) makefiles which are not in a source directory (for instance `pkg/MyFirstPackage/Makefile`) start with an upper-case ‘*M*’, while the source makefiles starts with a lower-case ‘*m*’.

If unsure, here is a command that could help you to check whether you did not forget any ones, to be executed from the root directory:

```
| find . -name '[Mm]akefile.??' | sed 's|^\.\/||;s|\...\$||' | uniq
```

You can also have a look at the `configure.scan` file generated by the `autoscan` command executed from the root directory.

The `E3D_wizard` script. The script `E3D_wizard` tries to automatize the previous (potentially tedious) procedure. To create the new project *MyProject* version `0.0` from the *E3D_Template-1.0* that you will have previously untarred, execute it from the `$E3D_PATH` directory as follow:

```
| E3D_wizard.sh -t E3D_Template-1.0 -P MyProject -V 0.0 -p MyFirstPackage
```

If you directly want to add an extra package to your new project, add the “`-a|--add`” option:

```
| E3D_wizard.sh -t E3D_Template-1.0 -P MyProject -V 0.0 -p MySecondPackage --add
```

2.3 Developing your project

Your development structure is now ready to use, but still pretty empty. This section explains how to add library and main sources to your project, but first some comments about the makefiles.

2.3.1 Some comments about the Makefiles

A *Makefile* is a handy-yet-cryptic file describing the proper way to compile large programs or groups of programs, what are the dependencies between the different parts of the project (libraries, includes, etc.), what are the targets and so on. It is assumed that you already know the basics regarding the makefiles, therefore we only point out the specificities of the *Euro3D-LCL* libraries.

- All the straight *Makefile* and *makefile* files of the project are *dynamically* generated from configuration files (i.e., directly from `[Mm]akefile.in` or indirectly from `Makefile.am`) by the appropriate commands. As a consequence, *none of them should ever be edited by hand*.
- The *Makefiles* (capital ‘*M*’) are rather technical, and you should not have to bother with them directly (except by editing the `Makefile.am` files as noted here and there).

- The only *real makefiles* you have to deal with are the ones in the `src/` and `lib/` directories, i.e., the ones starting with a lower-case ‘*m*’. They are generated from the associated *makefile.in*, which, as far as you are concerned, *are* the files to be edited.

Summary: Do not edit the *makefile* as it is automatically generated when the `configure` command is executed: any change you introduce will be overwritten in the process. Instead, edit the *makefile.in*.

- The *makefiles* we are dealing with are generally separated in 4 parts. We describe here the main structure, leaving the details for next sections.

Includes: some project-wide definitions are included from specific files stored in `add_defs/`:

```
topdir = @top_srcdir@
include $(topdir)/add_defs/makedefs
include $(topdir)/add_defs/makedefs.local
```

makedefs contains the low-level definitions – *and therefore should never be edited* – while *makedefs.local* is the appropriate place to set your own definitions¹.

Local definitions: some local variables are defined (see next sections). E.g., the space-separated list of executables in `src/makefile.in`:

```
EXE = $(E)/Program1 $(E)/Program2 $(E)/Program3
```

If a list gets too long to fit on a single line, use the ‘\’ character to extend the list on several lines, e.g.,

```
EXE = $(E)/Program1 $(E)/Program2 \
      $(E)/Program3
```

(note the required TAB at the beginning of the continued line).

The targets: the line starting with `OUT :` defines the targets of the *makefile*, e.g.,

```
OUT : dirbin dirobj $(EXE)
```

where *dirbin* and *dirobj* – defined in `add_defs/makedefs` – are the directories storing the executables and objects respectively.

The dependencies and compilation rules: these lines define on which files each target depends, and how to obtain it (see below for examples).

2.3.2 Local definitions

In order to make the package well integrated in the project, it is useful to declare some package-specific variables in the `add_defs/add_defs.local`:

```
MyFirstPackage_PATH = $(topdir)/pkg/MyFirstPackage
MyFirstPackage_INC = $(MyFirstPackage_PATH)/incl/example.h
MyFirstPackage_LIB = $(MyFirstPackage_PATH)/obj/libexample.a
```

These variables will be accessible from every *makefile*, and can therefore ease the “communication” between the different packages.

2.3.3 Adding library sources

Once your development tree is set up, the next step is to add routines and programs to your software packages. We start with library sources. In the following, we assume that you are working under the `lib/` directory of the `MyFirstPackage` package (i.e., `MyProject/pkg/MyFirstPackage/lib/`), but you could as well work in the `MyProject/pkg/lib/` for a project-wide library.

¹Potentially overriding the *makedefs* ones

1. Create your source files, e.g., *LibSource1.c* and *LibSource2.c*, according to the Euro3D-LCL standards (see [2] for examples). If you are setting up your project and using the E3D_Template-1.0 project, there is a template source file called *libexample.c*. We *strongly* advise you to use it as a starting point, as it is properly doxygen-tailored (see Sect. 3).

2. Edit the *makefile.in* file:

- In the line starting with *OBJ =*, list the name of the object files you want to create, prefixed with *\$(O)/* (the variable ‘*O*’ – defined in *add_defs/makedefs* – is the variable containing the path to the object directory, *./obj/* in our case). The line should then look like:

```
| OBJ = $(O)/LibSource1.o $(O)/LibSource2.o
```

- Give a name to the library (potentially libraries) you want to create by setting the *LIB* variable, e.g.,

```
| LIB = $(MyFirstPackage_LIB)
```

- After the *OUT : dirobj \$(LIB)* line stating that your target is the object directory and the library, you should add the appropriate dependency and compilation rules for each source file, e.g.,

```
| $(O)/LibSource1.o : LibSource1.c $(STAND_INC) $(project_INC) $(MyFirstPackage_INC)
|                               $(C_COMPILE)
```

The first line states that *LibSource1.o* depends on *LibSource1.c* (and include-files defined in *add_defs/add_defs* and *add_defs.local*), and the second one that it is obtained via the *C_COMPILE* command.

- Finally, add also the equivalent line for the library by itself:

```
| $(LIB) : $(OBJ)
|           $(AR) rv $(LIB) $(OBJ)
|           $(RANLIB) $@
```

3. See Sect. 2.4.2 for the compilation procedure.

2.3.4 Adding program sources

The procedure to follow to add a program into a software package is very similar to the one used for a library source described in previous section, the main change being the compilation rules to be set in *makefile.in*. We still assume that we are working in the *MyFirstPackage* software package, but this time in the *src/* directory (i.e., *MyProject/pkg/MyFirstPackage/src/*).

1. Create your source files, e.g., *Program1.c* and *Program2.c*, according to the Euro3D-LCL standards. Once again, the E3D_Template-1.0 project provides a template file called *E3D_template.c*. And once again we *strongly* advise you to use it as a starting point.

2. Edit the *makefile.in* file:

- In the line starting with *EXE =*, list the name of the executable files you want to create, prefixed with *\$(E)/* (the variable ‘*E*’ – defined in *add_defs/makedefs* – is the variable containing the path to the executable directory, *../user/bin/* in our case). The line should then look like:

```
| EXE = $(E)/Program1 $(E)/Program2
```

- After the *OUT : dirbin dirobj \$(EXE)* line stating that your target is the executable and object directories and the executables, you should add the appropriate dependency and compilation rules for each source file, e.g. (see above):

```
| $(O)/Program1.o : Program1.c $(STAND_INC) $(project_INC) $(MyFirstPackage_INC)
|                               $(C_COMPILE)
```

These lines state that *Program1.o* depends on *Program1.c* and that the compilation command is *C_COMPILE*.

- To get the executable from the object, add lines similar to the following ones:

```
$(E)/E3D_example : $(O)/E3D_example.o $(MyFirstPackage_LIB) $(IOLIB)
                 $(LD) $(LDF) -o $@ $< $(IOLIB) $(ADD_LIBS) $(MyFirstPackage_LIB)
                 $(STRIP) $@
```

Once again, the first line states the dependencies, the next one is the linking procedure, and the last one the stripping one. The $(LOCLIB)$ variable has been initially defined to contain the local libraries to be linked in.

3. See Sect. 2.4.2 for the compilation procedure.

2.3.5 Using include files

To create your include files according to the Euro3D-LCL standards, the E3D_Template-1.0 project provides a template file called *template.h*. Similarly to the libraries, the include files can be stored in two directories, depending on their intended scope:

- under `pkg/incl/` for project-wide includes (e.g., for global `#define`),
- under `incl/` of a given package for a package-wide include (e.g., for function prototyping).

However, both directories are accessible from every makefiles thanks to the `add_defs/makedefs` (slightly mis-named) variables `PKG_INC = ../../incl` and `USER_INC = ../incl`. Furthermore, one can use the definitions of `add_defs/makedefs.local`.

An `incl/` directory does not contain any *makefile* so it will not be processed when the command `make` is executed. However, its contents have to be fully integrated in the project structure. Therefore, the *Makefile.am* above it will contain the following line:

```
| EXTRA_DIST = incl
```

2.3.6 Adding a data directory

When developing a software package, you may need to store data and/or personal configuration files. For this, you can create a “data” directory which will fully be part of the project. In particular, it will be automatically saved in the *MyProject-0.0.tar.gz* project distribution file (see Sect. 2.4.3).

Similarly to the `incl` directories:

1. Create your data directory, e.g., `MyData/`, either directly under `pkg/` (project-wide dataset) or under a given package (package-wide dataset).
2. Edit the corresponding *Makefile.am* file, i.e., the one in the directory containing `MyData/`, and add the data directory to the `EXTRA_DIST` line (or create this line is inexistent):

```
| EXTRA_DIST = incl MyData
```

2.4 Using your project

Well, your development tree is all set up, you wrote a lot fancy code, now what to do? Compile, and distribute your project around!

2.4.1 Pre-compilation

Before compiling your project, you have to make sure that the dynamically generated makefiles are up to date with respect to the configuration file you may have edited.

Depending on the modifications you made, the following sets of commands should be executed from the root directory of the project:

- If you edited a *Makefile.am*, for instance after setting up a new project, or because you added a new directory to the project tree, or if you edited the file *configure.ac* – most probably to add a new *[Mm]akefile* to the *AC_CONFIG_FILES* list – do:

```
| aclocal
| autoconf
| autoheader
| automake --add-missing
```

or simply use the `./autogen.sh` script. This will generate a new set of *[Mm]akefile.in*, so go to next step.

- If you edited a *makefile.in* from the *src/* or *lib/* – e.g., to add new sources in these directories – or if they have been generated from updated configuration files, you need to recreate the real *[Mm]akefiles* with the following command:

```
| ./configure
```

If you want to compile your project in debug mode, so that you can easily debug your code with usual debuggers, the `./configure` command can be run with following option:

```
| ./configure --enable-debug
```

You also have the options `-with-optim=level` for optimization, and `-enable-profile` for profiling.

- If you just changed a source file, but not any of the configuration files, there is no pre-compilation required.

2.4.2 Compiling your project

Once your *[Mm]akefiles* are up-to-date, you can compile the whole project with a single command from the root directory:

```
| make
```

Note that executing this command will trigger the compilation of all the elements of your project that need it and only them. E.g., if you changed something in a library, it will recompile this library and all the programs linked against this library, as described in the dependancy rules. If you want a full compilation of the project, clean it first with command:

```
| make clean
```

before executing `make` again.

Warning: Always make sure you are in the root directory of your project when executing the `make` command. If this is not the case, it will run fine for files in the `lib/` directory but will not recompile the programs depending on the libraries you modified (while it *should* do it!).

2.4.3 Distributing your project

One very interesting feature available if you have followed all the steps described in the previous sections is that you can easily pack up your project and distribute it around. Indeed, while under the root directory of your project, the command:

```
| make dist
```

will create a complete distribution archive of your project. This will use the name (*MyProject*) and version (*0.0*) specified in the *AC_INIT* line of the *configure.ac* file (see item 6 of Sect. 2.2) to create a file called *MyProject-0.0.tar.gz*, which will contain all the source and include files (and data directory if needed) of your project, as well as all the needed configuration files.

To install your project on another computer where the Euro3D-LCL libraries are already installed, you just need this *MyProject-0.0.tar.gz* file, and the following commands, to install the very same project as the original one:

1. Copy the *MyProject-0.0.tar.gz* file into the root directory `$E3D_PATH` of the Euro3D-LCL distribution.
2. Untar the archive:

```
| tar xvzf MyProject-0.0.tar.gz
```

This will create a directory *MyProject-0.0/* containing all your project architecture and files.

3. In the *MyProject-0.0/* directory, execute the following command to compile your project:

```
| ./configure  
| make
```

That should be it!

3 Source documentation using doxygen

Warning: The structure of the doxygen headers is still under development and will probably change in the next months. Therefore, this section is still under development and must be considered as a (good) starting point.

The Euro3D-LCL source documentation (detailing how the code is structured, what are the algorithms used, etc.) is maintained under doxygen (cf. <http://www.doxygen.org/>), which can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in L^AT_EX) directly from the set of documented C-source files.

Warning: This document is not a doxygen tutorial: it just fix few standards regarding the possible comments to be added to the sources.

3.1 Standard documentation

We define here few standard documentation elements – namely the file and function documentation – that a developer should include in any new code. Since the doxygen comments are – according to some influent people – cryptic, and do not stand out clearly in the source, every doxygen comment block is delimited by two `/* === . . . === */` lines.

3.1.1 File documentation

Every source file (including .h files) should start with the header documenting the file by itself, as shown in Fig. 2. This standard header should include the following fields, some are mandatory (*), optional (o) and/or can appear more than once (>). Note that the `\copyright` and `\keywords` keys are non-standard doxygen fields.

`\file` (*) defines the filename,

`\copyright` (*) gives the copyright statement,

`\date` (o) precises the date of last modification. This is useful if it acts as a time-stamp when your text editor supports this feature (see Sect. 3.1.4),

`\author` (>) details the author(s), with one line per author,

`\keywords` (o) gives a list of keywords,

`\version` (>) recalls the history of the file, that is, for each version (e.g. as indicated in the `set_version` of the program), the date of release (dd/mm/yy), the developer's initials and the main modifications,

`\brief` (*) gives a *brief* description of the file content.

The following comment lines are considered as the *long* description of the file.

3.1.2 Function documentation

To document a function, you have to add the documentation *just before* the definition of the function (see Fig. 3), without any `#define` or global variables in between. The function header can include the following fields:

`\param` (>) describes each parameter of the function,

`\return` (*) describes the value returned by the function (if non *void*)

It is better to give a description of the function *without* the `\brief` field, which takes a lot of space in the L^AT_EX documentation.

doxygen file header

```

/* === Doxygen Comment ===== */
/*!
 * \file          fcq_param_errors.c
 * \copyright     (c) 2002 CRAL-Observatoire de Lyon / IPNL
 * \date         Tue Mar 5 16:45:07 2002
 * \author       Yannick Copin <y.copin@ipnl.in2p3.fr>
 * \keywords     Dynamics, FCQ
 * \version      1.0 06/04/01 YC creation (after \c fcq_losvd_errors)
 * \version      1.1 05/03/02 YC option \c -nosyst, debugging
 * \brief       Estimate errors on FCQ LOSVD parameters

Compute the errors on FCQ-estimated LOSVD parameters from Monte-Carlo
simulations.
*/
/* ===== */

```

Approximate output

fcq_param_errors.c

Estimate errors on FCQ LOSVD parameters

Compute the errors on FCQ-estimated LOSVD parameters from Monte-Carlo simulations.

Copyright:

(c) 2002 CRAL-Observatoire de Lyon / IPNL

Date:

Tue Mar 5 16:45:07 2002

Author:

Yannick Copin <y.copin@ipnl.in2p3.fr>

Keywords:

Dynamics, FCQ

Version:

1.0 06/04/01 YC creation (after fcq_losvd_errors)

1.1 05/03/02 YC option -nosyst, debugging

Figure 2: Example of a doxygen file header (*top*), and its corresponding output (*bottom*)

```

doxygen function header
/* === Doxygen Comment ===== */
/*!
 * Transformation in world coordinates for a frame
 *
 * \param frame [\c I] image structure
 * \param pixel_x [\c I] pixel along \f$x\f$ axis
 * \param pixel_y [\c I] pixel along \f$y\f$ axis
 * \param x [\c O] coordinate along \f$x\f$ axis
 * \param y [\c O] coordinate along \f$y\f$ axis
 * \return nothing, it's a \c void function!
 */
/* ===== */

```

```

void coord_frame(IMAGE2D *frame,int pixel_x,int pixel_y, float *x, float *y)
{
    *x = frame->startx + pixel_x * frame->stepx;
    *y = frame->starty + pixel_y * frame->stepy;
}

```

Approximate output

coord_frame

Transformation in world coordinates for a frame

Parameters:

frame [I] image structure
pixel_x [I] pixel along x axis
pixel_y [I] pixel along y axis
x [O] coordinate along x axis
y [O] coordinate along y axis

Returns:

nothing, it's a void function!

Figure 3: Example of a doxygen function header (followed by the function itself, *top*), and its corresponding output (*bottom*).

3.1.3 Other documentation

doxygen provides a lot of commands to let you document almost everything – global variables, algorithms, etc. – in your code (see [3]).

3.1.4 Emacs configuration

YC has developed a special IFU-mode for Emacs (to be updated for Euro3D-LCL). It automatically enforces the previous standard file and function *doxygen* documentation, and customizes the usual C-mode to take into account the IFU-specific types (e.g. `TIGERfile`). It is still buggy, but ask for it if you are an Emacs-addict.

3.2 *doxygen* configuration

The *doxygen* documentation is now fully integrated to (most of) Euro3D-LCL. As a consequence, the `doc/` directory of the Euro3D-LCL projects should contain a dedicated Doxygen directory. It will harbor the *makefile.in* to allow proper *doxygen* configuration and compilation.

After processing, the `doc/Doxygen` directory will have two subdirs:

- `html/` will contain the source documentation in HTML format (see *index.html*),
- `latex/` will contain the L^AT_EX version. It should have been compiled on the fly to produce the PDF-file *refman.pdf*.

Do not change anything in these directories, as they are generated automatically by *doxygen*. Instead, change the *doxygen*-comments in the C-code (`.h` and `.c` files), or the *doxygen* options through the `doc/Doxygen/Doxyfile` automatically generated and edited by *makefile.in*.

4 Writing the *User's Guide*

Of course, all these nice programs and libraries you developed in the Euro3D-LCL framework would not be all that nice if nobody knew how to use them. This is why you should (have to) write a complete *User's Guide* to your project.

In theory, it would possible to include the *User's Guide* to the doxygen source documentation (*Developer's Guide*). However, it appears that the two docs have a very different targeted audience, and end-users usually do not want to know about the nasty details of the code. Still, it is not always clear to where a given piece of information (e.g., an algorithm description) should go.

The E3D_Template provides a *User's Guide* template under doc/Manual/. This relies on the E3D L^AT_EX-style in doc/E3D/, where you can find the associated documentation (see [5]). The *Developer's Guide* that you have in your hands have been written with this style, and can be seen as a practical example (in doc/E3D_dev/).

A TODOs

You are more than welcome to participate to any of the following developments.

A.1 Template

- Go for a full autotools support² (never write a *makefile.in* anymore, and benefit from the whole machinery, including the yet-unavailable `distcheck` target).
- Add GUI support (but have to wait for a clear Euro3D GUI layout)
- doc: add TAGS support
- doc: add `doc/doxygen/tags` generic targets
- Doxygen: use *Doxyfile.in* (but difficult then to find the appropriate list in `AC_CONFIG_FILES`)
- Doxygen: add CVS/SVN support in file headers
- Port to CVS/SVN
- Finalize *E3D.el* Emacs-mode and deliver it
- *E3D_wizard*: see `autotoolset/acconfig`

A.2 Dev's Guide

- Proof-read
- Add a section about GNU-files *ChangeLog*, *README*, etc.
- Produce an index (of what?)
- FAQ (if enough feedback)

A.3 Misc.

- Euro3D-LCL: project *add_defs/makedefs* should not have to know about anything else than *IFU_io.h* (which should be touched when library included files are modified)

²By the way, this is not done yet because I don't know how to put the object files in `../obj` with `automake`. Help is welcome!