# Orientation Library

The documentation for Orientation Library 2.0.3
A collection of routines for orientation manipulation

3 April 2020

**Romain Quey**
École Nationale Supérieure des Mines de Saint-Étienne, France.

# Table of Contents

# Copying Conditions

The Orientation Library is "free software"; this means that everyone is free to use it and to redistribute in other free programs. The library is not in the public domain; it is copyrighted and there are conditions on its distribution. These conditions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of the software that they might get from you.

Specifically, we want to make sure that you have the right to share copies of programs that you are given which use the Orientation Library, that you receive their source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of any code which uses the Orientation Library, you must give the recipients all the rights that you have received. You must make sure that they, too, receive or can get the source code, both to the library and the code which uses it. And you must tell them their rights. This means that the library should not be redistributed in proprietary programs.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the Orientation Library – neither the routines nor the interactive program. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license for the Orientation Library are found in the GNU General Public License (see Appendix C [GNU General Public License], page 67). Further information about this license is available from the GNU Project webpage `http://www.gnu.org/copyleft/gpl-faq.html`.

The Orientation Library can be downloaded from `http://sourceforge.net/projects/orilib`. As the library evolves (new functionalities, bug fixes, manual improvements . . . ), you should check this site to get the latest version of the software. You should also consider subscribing to the mailing lists:

- orilib-announce: the "read-only" list for important news: new releases, bug fixes, etc.

  to subscribe, visit `https://lists.sourceforge.net/lists/listinfo/orilib-announce`.
- orilib-users: the list for users.

  Please send all questions, bug reports (including any data enabling to reproduce it), requests or any errors or omissions in this manual to this list. Feel free; feedback is very welcome.

  to subscribe, visit `https://lists.sourceforge.net/lists/listinfo/orilib-users`,
  to send a message, use `orilib-users@lists.sourceforge.net`.

Also, note that the library grows as per the author's needs (and time . . . ). If you think something is missing in the library, have a piece of code (and some explainations) for it, and want it to be available, you can contribute by sending it to the author at `rquey@users.sourceforge.net`.

If you use the Orientation Library, it would be appreciated that you mention it in your work. The recommended way is,
*R. Quey, Orientation Library: A collection of routines for orientation manipulation (Version 2.0.3), http://sourceforge.net/projects/orilib.*

# 1 Introduction

The Orientation Library is a collection of low- to high-level routines for rotation/orientation manipulation. It ranges from very general tools to others specific to materials science (crystal orientations). It has been written in ANSI C, and so is intended to compile on any system with a working ANSI C compiler. The library is primarily available as a (small) collection of source files. It also includes an interactive program that enables to run most of the routines, either manually or from a script. It is distributed under the GNU General Public License.

## 1.1 Why This Library?

Mathematical tools are needed to characterize rotational positions –*orientations*– in space. In this library, we primarily focus on lattice orientations in crystalline materials. Consequently, some of the provided tools are very general and some others are really specific to this purpose. While there are needs of such tools, there is a lack of a freely-available and well-documented library of routines. Moreover, there are plenty of books, articles and courses available, but, from reasoning to typing mistakes, they may contain erroneous results, which is a real problem for the end-user.
The Orientation Library was born from the collation of various documents on orientation manipulation, which are listed in Section 7.1 [References], page 59; it also includes new elements. It aims to get together the key-tools useful for orientation manipulation, in a reliable way. The useful notions, the implemented algorithms and the available routines are pedagogically documented. Example programs are also provided.

## 1.2 Routines Available in the Library

- Orientation description

  The most widely used descriptors are available, from Euler angles to quaternions. Functions make it possible to convert a descriptor into another.
- Orientation generation

  The library provides functions to generate orientations, either in the whole orientation space or about a particular orientation, as well as misorientations.
- Orientation calculation

  The library provides low- to high-level functions for orientation calculation, from, e.g., the calculation of an inverse rotation to disorientation calculation. The descriptors used are rotation matrix and quaternion.
- Orientation set

  This part includes functions specific to orientation sets, such as orientation averaging or orientation spread study.
- Orientation mapping

  The library provides functions specific to orientation mapping, from geometrical transformation to printing as an image.

To avoid conflicts, all exported function names have the prefix `ol_`. Macros and structures are uppercased and have the prefix `OL_`. The sole macro is `OL_VERSION` which is the library version (currently '`2.0.3`').

Angle values are expressed in *degrees* by default, but you can get it in *radians* by adding the suffix '`_rad`' to the function names whenever it makes sense. Also note that, when the input and output arguments of a function are of the same kind, say Euler angles, they may be described by the same variable without causing conflicts. For example, for function '`ol_e_rad2deg (`*`in`*`, `*`out`*`)`' which converts angles expressed in degrees, *`in`*, into radians, *`out`*, using '`ol_e_rad2deg (e, e)`' will just overwrite `e`.

## 1.3 Using the Library

### 1.3.1 Manual

This manual is maintained as a Texinfo manual. Here are the writing conventions used in the document:

- A command that can be typed in a terminal is printed like `this`;
- a program (or command) option is printed like `this`;
- The name of a variable is printed like `this`;
- A metasyntactic variable (i.e. something that stands for another piece of text) is printed like *this*;
- Literal examples are printed like '`this`';
- File names are printed like `this`.

Inverse trigonometric functions are used in the library. The usual arc sine, arc cosine and arc tangent functions have values in the units of degrees in the following ranges:

$$\begin{aligned} \operatorname{acos}(x) &\in [\ \ 0, 180] \\ \operatorname{asin}(x) &\in [-90,\ 90] \\ \operatorname{atan}(x) &\in [-90,\ 90] \end{aligned}$$

In addition, for the sake of convenience, the following mathematical functions are introduced:

- $\operatorname{sgn}(x)$: the sign of $x$.

$$\begin{aligned} &\text{if } x \geq 0, \quad \operatorname{sgn}(x) = \ \ 1 \\ &\text{if } x < 0, \quad \operatorname{sgn}(x) = -1 \end{aligned}$$

- $\operatorname{atan}_2(s, c)$: the angle whose cosine is $c$ and sine is $s$.

$$\operatorname{atan}_2(s, c) = \operatorname{sgn}(s) \operatorname{acos}(c)$$

  It has a value in $[-180, 180]$.

### 1.3.2 Source Files

For easy incorporation into programs, this library is primarily distributed as a small collection of files which is organized as this manual:

`ol_des.c, ol_gen.c, ol_cal.c, ol_set.c, ol_map.c, ut_4ol.c`
: These are the "main" files associated to the chapters of this manual, together with a "utility" file. For the sake of simplicity, you can include all of them into your project by default.

`ol_set_dep.c`
: This file comes along `ol_set.c`, but it contains some functions that require the `GSL` (they are indicated in the manual).

`ol_map_dep.c`
: This file comes along `ol_map.c`, but it contains some functions that require `libpng` (they are indicated in the manual).

`ut_4ol_dep.c`
: This file contains some "utility" functions that require the `GSL`. It is required if `ol_set_dep.c` is in use.

Each source file comes with a source code header file `.h` that must be included as soon as the source file is in use. For the sake of convenience, two more source code header files are provided: `ol_nodep.h`, which calls all the source code header file that do not require dependencies (`ol_des.h`, `ol_gen.h`, `ol_cal.h`, `ol_set.h`, `ol_map.h`, `ut_4ol.h`), and `ol_nodep.h` which call all source code header files.

The files that have dependencies – tagged `_dep` – contain functions that require tools not available in the standard libraries. These functions are quite specific and so may be useless for the user, that is why they are separated from the rest of the library.

Here are details on the dependencies that, when required, must be properly installed on your system:

- The GSL (GNU Scientific Library), a numerical library for C and C++ programmers.

  For Unix-type systems, see `http://www.gnu.org/software/gsl/`; for MS Windows, see `http://gnuwin32.sourceforge.net/packages/gsl.htm`.
- libpng, the official PNG reference library.

  For any system, see `http://www.libpng.org/pub/png`.

Furthermore, for easy modification, an hierarchical tree of (smaller) files comes along with these files. A small Shell script enables to build automatically the stand-alone files. The library files are to be used as any source code file of your project.

#### 1.3.2.1 An Example Program

The following program (available from directory `examples/ex0`) demonstrates the use of the library for converting Euler angles into a rotation matrix,

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"

int
main (void)
{
  double *  e = ol_e_alloc ();
  double ** g = ol_g_alloc ();

  printf ("Euler angles?\n");
  ol_e_fscanf (stdin, e);

  ol_e_g (e, g);

  printf ("Rotation matrix:\n");
  ol_g_fprintf (stdout, g, "%15.12f");

  ol_e_free (e);
  ol_g_free (g);

  return EXIT_SUCCESS;
}
```

#### 1.3.2.2 Compiling and Running

You have to compile the library source code files as any other source code file of your project. The typical commands with the GNU compiler `gcc` is,

```
$ gcc -Wall -ansi ex0.c ol/ol_des.c ol/ol_gen.c ol/ol_cal.c ol/ol_set.c\
  ol/ol_map.c ol/ut_4ol.c -lm
```

Note that you have to link with the mathematical library (option `-lm`).

Then, you can run the program and give input data,

```
$ ex0
Euler angles?
45 0 0
```

The output is shown below,

```
Rotation matrix:
 0.707106781187  0.707106781187  0.000000000000
-0.707106781187  0.707106781187  0.000000000000
 0.000000000000 -0.000000000000  1.000000000000
```

### 1.3.3 Interactive Program

The Orientation Library also includes a very simple interactive program that enables one to run the available routines. Executable versions are available for Unix-type systems (Intel, `libc version 2.19`) as well as 32-bit MS Windows.

The program header looks like this,

```
$ orilib

=============    o r i e n t a t i o n    l i b r a r y    =============
=============    p r o g r a m    v e r s i o n    2.0.3    =============

Copyright (C) 2007, 2008, 2014 Romain Quey
<http://sourceforge.net/projects/orilib>
This program comes with ABSOLUTELY NO WARRANTY; this is free software,
and you are welcome to redistribute it under certain conditions; for
details, type ‘license’.  To list all functions, type ‘listall’.  To
get help, type ‘help’.  To quit, type ‘quit’.
```

As can be seen above, the program includes some general-purpose commands which are self-explanatory: `license`, `listall`, `help` and `quit`. Note that the program itself includes few help and information. If you need further details about functions, their inputs and outputs, the best way it to refer to this manual.

To use a function, type its name then give the input data requested. The program will just provide the function output. Here is an example to convert Euler angles into rotation matrix,

```
ol_e_g                                          [input ]
e =                                             [output]
45 0 0                                          [input ]
g =                                             [output]
  0.707106781  0.707106781  0.000000000  [output]
 -0.707106781  0.707106781  0.000000000  [output]
  0.000000000 -0.000000000  1.000000000  [output]
```

After completion, the program waits for a new command.

There is a second way to use the program: give the function that you want to use as argument,

```
$ orilib ol_e_g
```

The program does not print any information, but just wait for the input data. When provided,

```
45 0 0                                          [input ]
```

the program simply gives the result,

```
  0.707106781  0.707106781  0.000000000  [output]
 -0.707106781  0.707106781  0.000000000  [output]
  0.000000000 -0.000000000  1.000000000  [output]
```

and quits.

This is particularly convenient for scripting, using pipes, etc. Here is a Shell example,

```
$ echo "45 0 0" | orilib ol_e_g
  0.707106781  0.707106781  0.000000000  [output]
 -0.707106781  0.707106781  0.000000000  [output]
  0.000000000 -0.000000000  1.000000000  [output]
```

# 2 Orientation Description

The functions described in this chapter are available from files `ol_des.[c,h]`. Due to dependencies, files `ol_cal.[c,h]` must also be included.

## 2.1 Reference and Crystal Coordinate Systems

To define an orientation, one must refer to coordinate systems: a *reference coordinate system attached to the specimen* $C_S$ and a *coordinate system attached to the crystal* $C_C$, see Figure 2.1. The reference coordinate system is usually aligned with important axes of the specimen; e.g. for a rolled product, the rolling, transverse and normal directions (RD, TD and ND, respectively). The crystal coordinate system is aligned with important axes of the crystal, e.g. for cubic crystals, the [100], [010] and [001] axes. Both coordinate systems are chosen to be orthonormal.

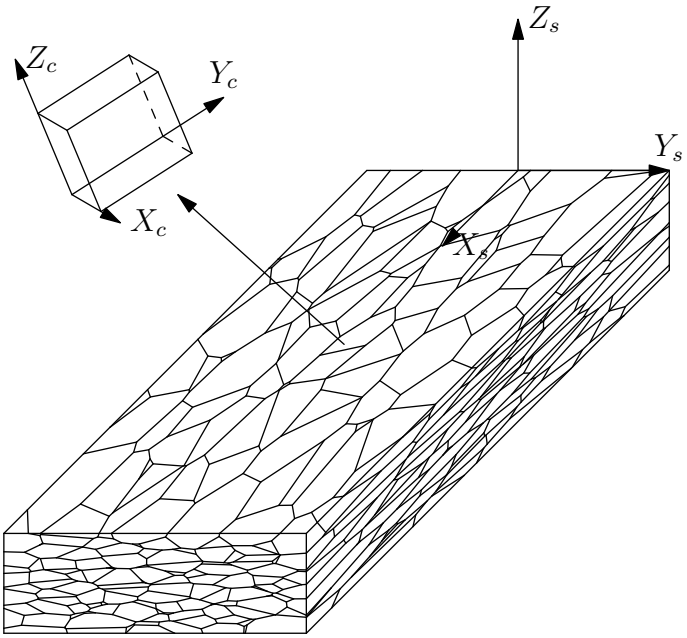



Figure 2.1: Reference coordinate system $C_s$ and crystal coordinate system $C_c$.

Usually there are several possibilities for the reference and crystal coordinate systems. The specimen may exhibit one of more planes of symmetry; e.g. at the middle of a rolled sheet, they are those of normal $X_s$ and $Y_s$ (orthorhombic symmetry). Consequently, they are several ways in which the reference coordinate system can be arranged. As for the crystal, this is related to crystal symmetry. For example, for cubic crystals, there are 24 ways in which the crystal coordinate system can be arranged – these are the so-called *crystallographically-related solutions*. Among all of these solutions, it is common to choose the one which is the closest to the reference coordinate system (minimal disorientation).

The *orientation of the crystal* is defined as the rotational position of the crystal coordinate system w.r.t. the reference coordinate system. In the three-dimensional space, it is described by *three independent variables*. There are different, but equivalent *orientation descriptors*, ranging from Euler angles to quaternions, see Figure 2.2. Each one can be more appropriate from case to case. Functions make it possible to convert a descriptor into another, either directly (when available) or through an intermediate descriptor.

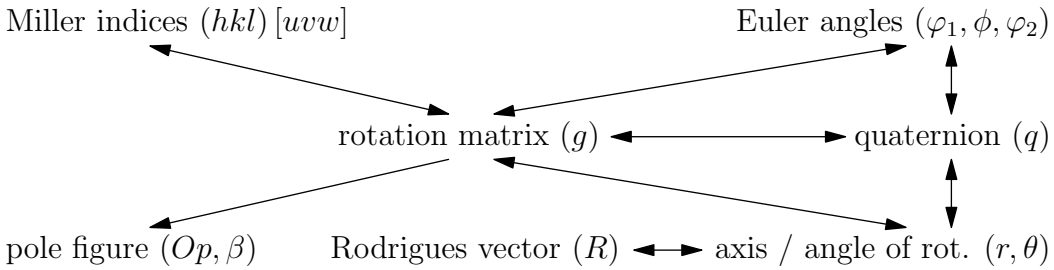



Figure 2.2: Available orientation descriptors and direct conversions (the indirect ones are not illustrated).

## 2.2 Rotation Matrix

The *rotation matrix* $g$ is defined as follows,

$$\begin{pmatrix} X_c \\ Y_c \\ Z_c \end{pmatrix} = \begin{pmatrix} g_{11} & g_{12} & g_{13} \\ g_{21} & g_{22} & g_{23} \\ g_{31} & g_{32} & g_{33} \end{pmatrix} \begin{pmatrix} X_s \\ Y_s \\ Z_s \end{pmatrix}$$

By construction, its rows contain the coordinates of the crystal coordinate system vectors in the reference coordinate system, and similarly, its columns contain the coordinates of the reference coordinate system vectors in the crystal coordinate system. Since the reference and crystal coordinate systems are orthonormal, $g$ is orthonormal and only three parameters are actually required to define it.

The main advantage of this descriptor is that it is relatively convenient for rotation calculations (inverse rotation, combination of rotations, etc.); its main drawback is that it contains nine parameters although only three are required.

### 2.2.1 Allocation

The rotation matrix $g$ is represented by a two-dimensional array, `g[i][j]` with `i` and `j` in $\{0, 1, 2\}$.

`double** ol_g_alloc` (*void*) [Function]
`void ol_g_free` (*double** g*) [Function]
The first routine creates a rotation matrix; it is initialized to identity.
The second frees a previously allocated rotation matrix *g*.

### 2.2.2 Initializing and Copying

`void ol_g_set_zero` (*double** g*) [Function]
This function sets a rotation matrix *g* to 0.

`void ol_g_set_id` (*double** g*) [Function]
This function sets a rotation matrix *g* to the identity rotation ($g_{ii} = 1$, $g_{ij} = 0$).

`void ol_g_set_this` (*double** g, double g11, double g12, double g13, double g21, double g22, double g23, double g31, double g32, double g33*) [Function]
This function sets a rotation matrix *g* using the terms given in argument.

`void ol_g_memcpy` (*double** g, double** g2*) [Function]
This function copies a rotation matrix *g* into a rotation matrix *g2*. *g2* must be preallocated.

### 2.2.3 Reading and Writing

`int ol_g_fscanf` (*FILE* stream, double** g*) [Function]
This routine reads formatted data from a stream *stream*[1] into a rotation matrix *g*. *g* must be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_g_fprintf` (*FILE* stream, double** g, char* format*) [Function]
This routine writes a rotation matrix *g* to the stream *stream*[2] using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

[1] '`stdin`' makes it possible to use the prompt.
[2] '`stdout`' makes it possible to use the prompt.

## 2.3 Euler Angles

The Euler angles, expressed in Bunge convention $(\varphi_1, \phi, \varphi_2)$, describe an orientation through three successive rotations about different axes. The angles are defined in the following ranges: $\varphi_1 \in [0, 360[$, $\phi \in [0, 180]$ and $\varphi_2 \in [0, 360[$.

Let '$\bullet'$', '$\bullet''$' and '$\bullet'''$' be the positions of vector '$\bullet$' after the first, second and third rotations, respectively. The rotations are (see Figure 2.3),

1. $\varphi_1$ about $Z_s$

   in such a way that ${X_s}'$ is normal to the plane containing vectors $Z_s$ and $Z_c$, and in the same sense than $Z_s \wedge Z_c$.

2. $\phi$ about ${X_s}'$

   in such a way that ${Z_s}''$ is coincident with $Z_c$.

3. $\varphi_2$ about ${Z_s}''$

   in such a way that ${X_s}'''$, ${Y_s}'''$ and ${Z_s}'''$ are coincident with $X_c$, $Y_c$ and $Z_c$.

It must be noted that, if $\phi = 0$, the first and third rotations are about the same axis $Z_s = {Z_s}''$, and consequently only $(\varphi_1 + \varphi_2)$ matters (not the individual values). This is the so-called *Euler space degeneracy*. Similarly, if $\phi = 180$, only $(\varphi_1 - \varphi_2)$ matters. In these cases, by convention, $\varphi_2 = 0$.

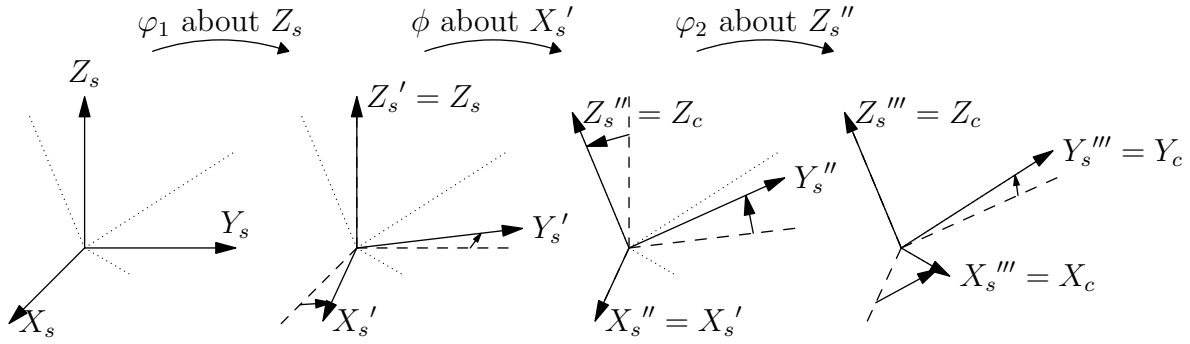



Figure 2.3: Description of the rotation from the reference coordinate system to the crystal coordinate system through Euler angles. Solid line: current coordinate system, dash line: previous coordinate system, dot line: final coordinate system.

Conversions of Euler angles into rotation matrix, and vice-versa, are,

$$\begin{cases} g_{11} = & \cos\varphi_1 \cos\varphi_2 - \sin\varphi_1 \sin\varphi_2 \cos\phi \\ g_{12} = & \sin\varphi_1 \cos\varphi_2 + \cos\varphi_1 \sin\varphi_2 \cos\phi \\ g_{13} = & \sin\varphi_2 \sin\phi \\ g_{21} = & -\cos\varphi_1 \sin\varphi_2 - \sin\varphi_1 \cos\varphi_2 \cos\phi \\ g_{22} = & -\sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\phi \\ g_{23} = & \cos\varphi_2 \sin\phi \\ g_{31} = & \sin\varphi_1 \sin\phi \\ g_{32} = & -\cos\varphi_1 \sin\phi \\ g_{33} = & \cos\phi \end{cases}$$

and,

$$\phi = \text{acos}\ (g_{33})$$

$$\text{if } \begin{cases} \phi \neq 0 \\ \phi \neq 180 \end{cases}, \quad \begin{cases} \varphi_1 = \text{atan}_2\,(g_{31}, -g_{32}) \\ \varphi_2 = \text{atan}_2\,(g_{13}, \quad g_{23}) \end{cases}$$

$$\text{if } \begin{cases} \phi = 0 \text{ or} \\ \phi = 180 \end{cases}, \quad \begin{cases} \varphi_1 = \text{atan}_2\,(g_{12}, g_{11}) \\ \varphi_2 = 0 \end{cases} \quad \text{by convention.}$$

For conversion of Euler angles into quaternion, and vice-versa, see Section 2.7 [Quaternion], page 23.

### 2.3.1 Allocation

The Euler angles $e$ are represented by a one-dimensional array, `e[i]` with `i` in $\{0, 1, 2\}$.

`double* ol_e_alloc` (*void*) [Function]
`void ol_e_free` (*double* e*) [Function]
The first routine creates Euler angles; they are initialized to $(0, 0, 0)$.
The second frees a previously allocated Euler angles $e$.

### 2.3.2 Initializing and Copying

`void ol_e_set_zero` (*double* e*) [Function]
This function sets Euler angles $e$ to 0.

`void ol_e_set_id` (*double* e*) [Function]
This function sets Euler angles $e$ to the identity rotation $(0, 0, 0)$.

`void ol_e_set_this` (*double* e*, *double phi1*, *double phi*, *double phi2*) [Function]
This function sets Euler angles $e$ using the terms given in argument.

`void ol_e_memcpy` (*double* e*, *double* e2*) [Function]
This function copies Euler angles $e$ into Euler angles *e2*. *e2* must be preallocated.

### 2.3.3 Conversions

`void ol_g_e` (*double** g*, *double* e*) [Function]
This routine converts a rotation matrix $g$ into Euler angles $e$.

`void ol_e_g` (*double* e*, *double** g*) [Function]
This routine converts Euler angles $e$ into a rotation matrix $g$.

`void ol_e_e` (*double* e*, *double* e2*) [Function]
This routine converts Euler angles $e$ into Euler angles *e2* so that ($\varphi_1 \in [0, 360[$, $\phi \in [0, 180]$ and $\varphi_2 \in [0, 360[$).

`void ol_e_deg2rad` (*double* e*, *double* e2*) [Function]
This routine converts Euler angles $e$ expressed in degrees into Euler angles *e2* expressed in radians.

`void ol_e_rad2deg` (*double* e*, *double* e2*) [Function]
This routine converts Euler angles $e$ expressed in radians into Euler angles *e2* expressed in degrees.

Some functions of indirect conversion are also provided,

`void ol_e_rtheta` (*double* e*, *double* r*, *double* &theta*) [Function]
`void ol_e_R` (*double* e*, *double* R*) [Function]
`void ol_e_q` (*double* e*, *double* q*) [Function]
These routines convert Euler angles $e$ into a axis / angle of rotation *(r, theta)*, a Rodrigues vector $R$ or a quaternion $q$, respectively.

### 2.3.4 Reading and Writing

`int ol_e_fscanf` (*FILE** `stream`, *double** `e`) [Function]
This routine reads formatted data from the stream *stream* into Euler angles *e*. *e* must be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_e_fprintf` (*FILE** `stream`, *double** `e`, *char** `format`) [Function]
This routine writes Euler angles *e* to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.4 Miller Indices

In metallurgy, (ideal) orientations are commonly described through the Miller indices notation $(hkl)\,[uvw]$, where $(hkl)$ is the plane coincident with plane $X_s - Y_s$ and $[uvw]$ is the direction parallel to $X_s$. For a rolled specimen, they are the rolling plane and direction.

The conversion of Miller indices into rotation matrix is,

$$g = \begin{pmatrix} u/n & (kw-lv)/mn & h/m \\ v/n & (lu-hw)/mn & k/m \\ w/n & (hv-ku)/mn & l/m \end{pmatrix} \text{ with } \begin{cases} m = \sqrt{h^2+k^2+l^2} \\ n = \sqrt{u^2+v^2+w^2} \end{cases}$$

The conversion of rotation matrix into Miller indices is not straightforward. $(hkl)$ and $[uvw]$ are obtained from the last and first columns of $g$, respectively. They are calculated by multiplying the matrix column by a suitable factor and then rounding it to obtain whole numbers. This leads to several choices for the Miller indices $(hkl)[uvw]$, which, on the one hand, should be as low as possible, and on the other hand, should be an orientation as close as possible to the original orientation – let $\theta$ be the disorientation angle. Moreover, $(hkl)$ and $[uvw]$ must be orthogonal vectors.

Among all possible solutions, in the library,

- Miller indices are chosen to be lower or equal than a given limit in absolute value (usually 9; specified by the user);
- A maximal amount of possible Miller indices is chosen (e.g. 10; specified by the user);
- the Miller indices are chosen following the "quality" criterion,

$$m_q = (1-\alpha_q)\underbrace{(|h|+|k|+|l|+|u|+|v|+|w|)}_{\Sigma} + \alpha_q\theta \text{ minimum}$$

  where $\alpha_q = 1/2$ by default.

For example, the following orientation (Euler angles: $(0, 50, 0)$),

$$g = \begin{pmatrix} 1.000000 & 0.000000 & 0.000000 \\ 0.000000 & 0.642788 & 0.766044 \\ 0.000000 & -0.766044 & 0.642788 \end{pmatrix}$$

leads to the Miller indices listed in Table 2.1.

| **Miller indices** | $\theta$ | $\Sigma$ | $m_q$ |
|---|---|---|---|
| $\{011\}\langle100\rangle$ | 5.00 | 3 | 4.00 |
| $\{043\}\langle100\rangle$ | 3.13 | 8 | 5.57 |
| $\{054\}\langle100\rangle$ | 1.34 | 10 | 5.67 |
| $\{065\}\langle100\rangle$ | 0.19 | 12 | 6.09 |
| $\{032\}\langle100\rangle$ | 6.31 | 6 | 6.15 |
| $\{076\}\langle100\rangle$ | 0.60 | 14 | 7.30 |
| $\{087\}\langle100\rangle$ | 1.19 | 16 | 8.59 |
| $\{021\}\langle100\rangle$ | 13.43 | 4 | 8.71 |
| $\{075\}\langle100\rangle$ | 4.46 | 13 | 8.73 |
| $\{097\}\langle100\rangle$ | 2.13 | 17 | 9.56 |

Table 2.1: Miller indices obtained from a rotation matrix (Euler angles: $(0, 50, 0)$). See the body of the text for the definitions of $\theta$, $\Sigma$ and $m_q$.

### 2.4.1 Allocation

The Miller indices $(hkl)\,[uvw]$ are represented by a one-dimensional integer array, `m[i]` with `i` in $\{0, ..., 5\}$.

`int* ol_m_alloc` (*void*) [Function]
`void ol_m_free` (*int* `m`*) [Function]
The first routine creates Miller indices; they are initialized to $(001)\,[100]$.
The second frees the previously allocated Miller indices *m*.

### 2.4.2 Initializing and Copying

`void ol_m_set_zero` (*int* `m`*) [Function]
This function sets Miller indices *m* to 0.

`void ol_m_set_id` (*int* `m`*) [Function]
This function sets Miller indices *m* to the identity rotation $(001)\,[100]$.

`void ol_m_set_this` (*int* `m`, *int* `h`, *int* `k`, *int* `l`, *int* `u`, *int* `v`, *int* `w`) [Function]
This function sets Miller indices *m* using the terms given in argument.

`void ol_m_memcpy` (*int* `m`, *int* `m2`) [Function]
This function copies Miller indices *m* into Miller indices *m2*. *m2* must be preallocated.

### 2.4.3 Conversions

`void ol_m_g` (*int* `m`, *double*** `g`) [Function]
This routine converts Miller indices *m* into a rotation matrix *g*.

`void ol_g_m` (*double*** `g`, *int* `maxindex`, *int* `maxresqty`, *int*** `allm`, *double** `mq`, *int** `&resqty`) [Function]
`void ol_g_m_quality` (*double*** `g`, *int* `maxindex`, *int* `maxresqty`, *double* `alphaq`, *int*** `allm`, *double** `mq`, *int** `&resqty`) [Function]
These routines convert a rotation matrix into Miller indices. *g* is the rotation matrix, *maxindex* is the maximal value for the indices, *maxresqty* is the maximum number of Miller indices solutions to provide, *allm* contains the set of Miller indices provided, *mq* are their "qualities" $m_q$. *resqty* is their quantity ($1 \leq resqty \leq maxresqty$). `ol_g_m_quality` makes it possible to tune the quality expression by specifying $\alpha_q$ ($\alpha_q = 1/2$ by default). *allm* and *mq* must be preallocated.

Some functions of indirect conversion are also provided,

`void ol_m_e` (*int* `m`, *double** `e`) [Function]
`void ol_m_rtheta` (*int* `m`, *double** `r`, *double** *&*`theta`) [Function]
`void ol_m_R` (*int* `m`, *double** `R`) [Function]
`void ol_m_q` (*int* `m`, *double** `q`) [Function]
These routines convert Miller indices *m* into Euler angles *e*, an axis / angle of rotation *(r, theta)*, a Rodrigues vector *R* or a quaternion *q*, respectively.

### 2.4.4 Reading and Writing

`int ol_m_fscanf` (*FILE** `stream`, *int** `m`) [Function]
This routine reads formatted data from the stream *stream* into Miller indices *m*. *m* must be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_m_fprintf` (*FILE** `stream`, *int** `m`, *char** `format`) [Function]
This routine writes Miller indices *m* to the stream *stream* using the format specifier *format*, which should be the `%d` format. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.5 Axis/Angle of Rotation

It is possible to bring the reference coordinate system to the crystal coordinate system by a rotation of angle $\theta$ about a particular axis $r$. The angle $\theta$ is defined in $[0, 180]$, see Figure 2.4.

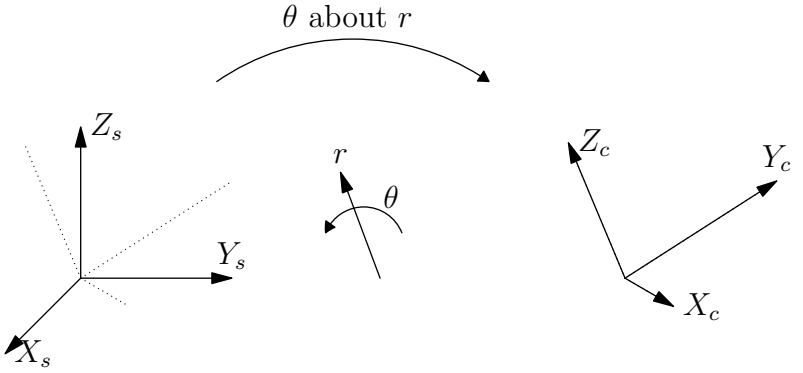



Figure 2.4: Description of the rotation from the reference coordinate system to the crystal coordinate system through the axis / angle of rotation.

Conversions of axis / angle of rotation into rotation matrix, and vice-versa, are,

$$\begin{cases} g_{11} = r_1 r_1 (1 - \cos\theta) + \cos\theta \\ g_{12} = r_1 r_2 (1 - \cos\theta) + r_3 \sin\theta \\ g_{13} = r_1 r_3 (1 - \cos\theta) - r_2 \sin\theta \\ g_{21} = r_2 r_1 (1 - \cos\theta) - r_3 \sin\theta \\ g_{22} = r_2 r_2 (1 - \cos\theta) + \cos\theta \\ g_{23} = r_2 r_3 (1 - \cos\theta) + r_1 \sin\theta \\ g_{31} = r_3 r_1 (1 - \cos\theta) + r_2 \sin\theta \\ g_{32} = r_3 r_2 (1 - \cos\theta) - r_1 \sin\theta \\ g_{33} = r_3 r_3 (1 - \cos\theta) + \cos\theta \end{cases}$$

and,

$$\theta = \operatorname{acos} \frac{g_{11} + g_{22} + g_{33} - 1}{2}$$

$$\text{if } \begin{cases} \theta \neq 180 \\ \theta \neq 0 \end{cases}, \quad \begin{cases} r_1 = (g_{23} - g_{32}) / (2\sin\theta) \\ r_2 = (g_{31} - g_{13}) / (2\sin\theta) \\ r_3 = (g_{12} - g_{21}) / (2\sin\theta) \end{cases}$$

$$\text{if } \theta = 180, \quad \begin{cases} r_1 = \pm\sqrt{(g_{11}+1)/2} \\ r_2 = \pm\sqrt{(g_{22}+1)/2} \\ r_3 = \pm\sqrt{(g_{33}+1)/2} \end{cases} \text{ with } \left| \begin{array}{l} m \,|\, \forall\, i \in \{1,2,3\},\ |r_m| \geq |r_i|, \\ \begin{cases} r_m > 0 \text{ by convention.} \\ \forall i \neq m,\ \operatorname{sgn}(r_i) = \operatorname{sgn}(g_{im}) \end{cases} \end{array} \right.$$

$$\text{if } \theta = 0, \quad \begin{cases} r_1 = 1 \\ r_2 = 0 \quad \text{by convention.} \\ r_3 = 0 \end{cases}$$

### 2.5.1 Allocation

The angle of rotation is represented by a real number. The axis of rotation $r$ is represented by a one-dimensional array, `r[i]` with `i` in $\{0, 1, 2\}$.

`double* ol_r_alloc` (*void*) [Function]
`void ol_r_free` (*double* `r`*) [Function]
The first routine creates an axis of rotation; it is initialized to $(1, 0, 0)$.
The second frees a previously allocated axis of rotation $r$.

### 2.5.2 Initializing and Copying

`void ol_r_set_zero` (*double* `r`*) [Function]
This function sets an axis of rotation $r$ to 0.

`void ol_r_set_id` (*double* `r`*) [Function]
This function sets an axis of rotation $r$ to $(1, 0, 0)$.

`void ol_r_set_this` (*double* `r`*, double* `r1`*, double* `r2`*, double* `r3`*) [Function]
This function sets an axis of rotation $r$ using the terms given in argument.

`void ol_rtheta_set_zero` (*double* `r`*, double* *&*`theta`*) [Function]
This function sets an axis / angle of rotation ($r$, *theta*) to 0.

`void ol_rtheta_set_id` (*double* `r`*, double* *&*`theta`*) [Function]
This function sets an axis / angle of rotation ($r$, *theta*) to the identity rotation ($r = (1, 0, 0)$, $theta = 0$).

`void ol_rtheta_set_this` (*double* `r`*, double* *&*`theta`*, double* `r1`*, double* `r2`*, double* `r3`*, double* `thetaval`*) [Function]
This function sets an axis / angle of rotation ($r$, *theta*) using the terms given in argument.

`void ol_r_memcpy` (*double* `r`*, double* `r2`*) [Function]
This function copies an axis of rotation $r$ into an axis of rotation *r2*. *r2* must be preallocated.

`void ol_rtheta_memcpy` (*double* `r`*, double* `theta`*, double* `r2`*, double* *&*`theta2`*) [Function]
This function copies an axis / angle of rotation ($r$, *theta*) into an axis / angle of rotation (*r2*, *theta2*). *r2* must be preallocated.

### 2.5.3 Conversions

`void ol_g_rtheta` (*double*** `g`*, double* `r`*, double* *&*`theta`*) [Function]
This routine converts a rotation matrix $g$ into an axis / angle of rotation ($r$, *theta*).

`void ol_g_theta` (*double*** `g`*, double* *&*`theta`*) [Function]
This routine provides the angle of rotation *theta* associated to a rotation matrix $g$.

`void ol_rtheta_g` (*double* `r`*, double* `theta`*, double*** `g`*) [Function]
This routine converts an axis / angle of rotation ($r$, *theta*) into a rotation matrix $g$.

`void ol_rtheta_rtheta` (*double* `r`*, double* `theta`*, double* `r2`*, double* *&*`theta2`*) [Function]
This routine converts an axis / angle of rotation ($r$, *theta*) into an axis / angle of rotation (*r2*, *theta2*) with $theta2 \in [0, 180]$.

`void ol_theta_deg2rad` (*double* `theta`, *double** `&theta2`) [Function]
This function converts an angle *theta* expressed in degrees into an angle *theta2* expressed in radians.

`void ol_theta_rad2deg` (*double* `theta`, *double** `&theta2`) [Function]
This function converts an angle *theta* expressed in radians into an angle *theta2* expressed in degrees.

Some functions of indirect conversion are also provided,

`void ol_rtheta_e` (*double** `r`, *double* `theta`, *double** `e`) [Function]
`void ol_rtheta_R` (*double** `r`, *double* `theta`, *double** `R`) [Function]
`void ol_rtheta_q` (*double** `r`, *double* `theta`, *double** `q`) [Function]
These routines convert an axis / angle of rotation *(r, theta)* into Euler angles *e*, a Rodrigues vector *R* or a quaternion *q*, respectively.

### 2.5.4 Reading and Writing

`int ol_rtheta_fscanf` (*FILE** `stream`, *double** `r`, *double** `&theta`) [Function]
`int ol_r_fscanf` (*FILE** `stream`, *double** `r`) [Function]
`int ol_theta_fscanf` (*FILE** `stream`, *double** `&theta`) [Function]
These routines read formatted data from the stream *stream* into an axis / angle of rotation (*r*, *theta*), an axis *r* or an angle *theta*, respectively. When in use, *r* must be preallocated. The functions return `0` for success and `EOF` if there was a problem reading from *stream*.

`int ol_rtheta_fprintf` (*FILE** `stream`, *double** `r`, *double* `theta`, *char** `format`) [Function]
`int ol_r_fprintf` (*FILE** `stream`, *double** `r`, *char** `format`) [Function]
`int ol_theta_fprintf` (*FILE** `stream`, *double* `theta`, *char** `format`) [Function]
These routines write an axis / angle of rotation (*r*, *theta*), an axis *r* or an angle *theta*, respectively, to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The functions return a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.6 Rodrigues Vector

The Rodrigues vector representation $R$ derives from the axis / angle of rotation representation. It consists in a three-dimensional vector. The rotation axis gives the vector direction and the rotation angle its magnitude.

Conversions of axis / angle of rotation into Rodrigues vector, and vice-versa, are,

$$R = \tan(\theta/2)\, r$$

and,

$$\theta = 2 \operatorname{atan} |R|$$

$$\text{if } \theta \neq 0, \quad r = R / \tan(\theta/2)$$

$$\text{if } \theta = 0, \quad \begin{cases} r_1 = 1 \\ r_2 = 0 \\ r_3 = 0 \end{cases} \text{ by convention.}$$

Conversions of rotation matrix into Rodrigues vector, and vice-versa, can be easily made via the axis / angle of rotation.

For conversion of Rodrigues vector into quaternion, and vice-versa, see Section 2.7 [Quaternion], page 23.

### 2.6.1 Allocation

The Rodrigues vector $R$ is represented by a one-dimensional array, `R[i]` with `i` in $\{0, 1, 2\}$.

`double* ol_R_alloc` (*void*) [Function]
`void ol_R_free` (*double* `R`*) [Function]
The first routine creates a Rodrigues vector; it is initialized to $(0, 0, 0)$.
The second frees a previously allocated Rodrigues vector $R$.

### 2.6.2 Initializing and Copying

`void ol_R_set_zero` (*double* `R`*) [Function]
This function sets a Rodrigues vector $R$ to 0.

`void ol_R_set_id` (*double* `R`*) [Function]
This function sets a Rodrigues vector $R$ to the identity rotation $(0, 0, 0)$.

`void ol_R_set_this` (*double* `R`*, double* `R1`*, double* `R2`*, double* `R3`*) [Function]
This function sets a Rodrigues vector $R$ using the terms given in argument.

`void ol_R_memcpy` (*double* `R`*, double* `R2`*) [Function]
This function copies a Rodrigues vector $R$ into a Rodrigues vector *R2*. *R2* must be preallocated.

### 2.6.3 Conversions

`void ol_rtheta_R` (*double* `r`*, double* `theta`*, double* `R`*) [Function]
This routine converts an axis / angle of rotation (*r*, *theta*) into a Rodrigues vector $R$.

`void ol_R_rtheta` (*double* `R`*, double* `r`*, double* `&theta`*) [Function]
This routine converts a Rodrigues vector $R$ into an axis / angle of rotation (*r*, *theta*).

`void ol_g_R` (*double** `g`, *double** `R`) [Function]
This routine converts a rotation matrix $g$ into a Rodrigues vector $R$.

`void ol_R_g` (*double** `R`, *double** `g`) [Function]
This routine converts a Rodrigues vector $R$ into a rotation matrix $g$.

Some functions of indirect conversion are also provided,

`void ol_R_e` (*double** `R`, *double** `e`) [Function]
`void ol_R_q` (*double** `R`, *double** `q`) [Function]
These routines convert a Rodrigues vector $R$ into Euler angles $e$ or a quaternion $q$, respectively.

### 2.6.4 Reading and Writing

`int ol_R_fscanf` (*FILE** `stream`, *double** `R`) [Function]
This routine reads formatted data from the stream *stream* into a Rodrigues vector $R$. $R$ must be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_R_fprintf` (*FILE** `stream`, *double** `R`, *char** `format`) [Function]
This routine writes a Rodrigues vector $R$ to the stream *stream*, using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.7 Quaternion

The quaternion representation derives from the axis / angle of rotation representation. A quaternion $q = (\rho, \lambda, \mu, \nu)$ consists of a scalar $\rho$ and a vector $(\lambda, \mu, \nu)$. To describe rotations, *unit quaternions* are used: $|q| = \sqrt{\rho^2 + \lambda^2 + \mu^2 + \nu^2} = 1$.

Conversions of axis / angle of rotation into quaternion, and vice-versa, are,

$$\begin{cases} \rho = & \cos(\theta/2) \\ \lambda = r_1 \sin(\theta/2) \\ \mu = r_2 \sin(\theta/2) \\ \nu = r_3 \sin(\theta/2) \end{cases}$$

and,

$$\theta \ = 2 \operatorname{acos}(\rho)$$

$$\text{if } \theta \neq 0, \quad \begin{cases} r_1 = \lambda / \sin(\theta/2) \\ r_2 = \mu / \sin(\theta/2) \\ r_3 = \nu / \sin(\theta/2) \end{cases}$$

$$\text{if } \theta = 0, \quad \begin{cases} r_1 = 1 \\ r_2 = 0 \quad \text{by convention.} \\ r_3 = 0 \end{cases}$$

Direct conversion of quaternion into rotation matrix, and vice-versa, are,

$$\begin{cases} g_{11} = \rho^2 + \lambda^2 - \mu^2 - \nu^2 \\ g_{12} = 2(\lambda\mu + \rho\nu) \\ g_{13} = 2(\lambda\nu - \rho\mu) \\ g_{21} = 2(\lambda\mu - \rho\nu) \\ g_{22} = \rho^2 - \lambda^2 + \mu^2 - \nu^2 \\ g_{23} = 2(\mu\nu + \rho\lambda) \\ g_{31} = 2(\lambda\nu + \rho\mu) \\ g_{32} = 2(\mu\nu - \rho\lambda) \\ g_{33} = \rho^2 - \lambda^2 - \mu^2 + \nu^2 \end{cases}$$

and,

$$\rho = \sqrt{g_{11} + g_{22} + g_{33} + 1}/2$$

$$\text{if } \rho \neq 0, \quad \begin{cases} \lambda = (g_{23} - g_{32}) / (4\rho) \\ \mu = (g_{31} - g_{13}) / (4\rho) \\ \nu = (g_{12} - g_{21}) / (4\rho) \end{cases}$$

$$\text{if } \rho = 0, \quad \begin{cases} \lambda = \pm\sqrt{(g_{11} + 1)/2} \\ \mu = \pm\sqrt{(g_{22} + 1)/2} \\ \nu = \pm\sqrt{(g_{33} + 1)/2} \end{cases} \quad \text{with} \quad \left| \begin{array}{l} q_1 = \lambda,\ q_2 = \mu,\ q_3 = \nu \\ m \,|\, \forall\, i \in \{1,2,3\},\ |q_m| \geq |q_i|, \\ \begin{cases} q_m > 0 \text{ by convention} \\ \forall i \neq m,\ \operatorname{sgn}(q_i) = \operatorname{sgn}(g_{im}) \end{cases} \end{array} \right.$$

Direct conversion of quaternion into Euler angles, and vice-versa, are,

$$\begin{cases} \rho = \cos(\phi/2) \cos[(\varphi_1 + \varphi_2)/2] \\ \lambda = \sin(\phi/2) \cos[(\varphi_1 - \varphi_2)/2] \\ \mu = \sin(\phi/2) \sin[(\varphi_1 - \varphi_2)/2] \\ \nu = \cos(\phi/2) \sin[(\varphi_1 + \varphi_2)/2] \end{cases}$$

and,

$$\phi = 2\,\mathrm{atan}_2\left(\sqrt{\lambda^2+\mu^2}, \sqrt{\rho^2+\nu^2}\right)$$

$$\text{if } \begin{cases} \phi \neq 0 \\ \phi \neq 180 \end{cases}, \quad \begin{cases} \varphi_1 = \mathrm{atan}_2(\nu,\rho) + \mathrm{atan}_2(\mu,\lambda) \\ \varphi_2 = \mathrm{atan}_2(\nu,\rho) - \mathrm{atan}_2(\mu,\lambda) \end{cases}$$

$$\text{if } \phi = 0, \quad \begin{cases} \varphi_1 = 2\,\mathrm{arctan}_2(\nu,\,\rho) \\ \varphi_2 = 0 \end{cases} \quad \text{by convention.}$$

$$\text{if } \phi = 180, \quad \begin{cases} \varphi_1 = 2\,\mathrm{arctan}_2(\mu,\,\lambda) \\ \varphi_2 = 0 \end{cases} \quad \text{by convention.}$$

Conversions of quaternion into Rodrigues vector, and vice-versa, can be easily made via the axis / angle of rotation.

Moreover, it can be noticed that quaternions $q$ and $-q$ describe the same orientation. Usually, *positive quaternions* are used: the first non-zero term is positive.

### 2.7.1 Allocation

The quaternion $q$ is represented by a one-dimensional array, `q[i]` with `i` in $\{0, ..., 3\}$.

`double* ol_q_alloc` (*void*) [Function]
`void ol_q_free` (*double* q*) [Function]
The first routine creates a quaternion; it is initialized to $(1, 0, 0, 0)$.
The second frees a previously allocated quaternion $q$.

### 2.7.2 Initializing and Copying

`void ol_q_set_zero` (*double* q*) [Function]
This function sets a quaternion $q$ to 0.

`void ol_q_set_id` (*double* q*) [Function]
This function sets a quaternion $q$ to the identity rotation $(1, 0, 0, 0)$.

`void ol_q_set_this` (*double* q*, *double* rho*, *double* lambda*, *double* mu*, *double* nu*) [Function]
This function sets a quaternion $q$ using the terms given in argument.

`void ol_q_memcpy` (*double* q*, *double* q2*) [Function]
This function copies a quaternion $q$ into a quaternion *q2*. *q2* must be preallocated.

### 2.7.3 Conversions

`void ol_rtheta_q` (*double* r*, *double theta*, *double* q*) [Function]
This routine converts an axis / angle of rotation ($r$, *theta*) into a quaternion $q$.

`void ol_q_rtheta` (*double* q*, *double* r*, *double* &theta*) [Function]
This routine converts a quaternion $q$ into an axis / angle of rotation ($r$, *theta*).

`void ol_q_theta` (*double* q*, *double* &theta*) [Function]
This routine provides the angle of rotation *theta* associated to a quaternion $q$.

`void ol_g_q` (*double** g*, *double* q*) [Function]
This routine converts a rotation matrix $g$ into a quaternion $q$.

`void ol_q_g` (*double* q*, *double** g*) [Function]
This routine converts a quaternion $q$ into a rotation matrix $g$.

`void ol_e_q` (*double** `e`, *double** `q`) [Function]
This routine converts Euler angles $e$ into a quaternion $q$.

`void ol_q_e` (*double** `q`, *double** `e`) [Function]
This routine converts a quaternion $q$ into Euler angles $e$.

`void ol_R_q` (*double** `R`, *double** `q`) [Function]
This routine converts a Rodrigues vector $R$ into a quaternion $q$.

`void ol_q_R` (*double** `q`, *double** `R`) [Function]
This routine converts a quaternion $q$ into a Rodrigues vector $R$.

`void ol_q_q` (*double** `q`, *double** `q2`) [Function]
This routine converts a quaternion $q$ into a positive quaternion $q2$.

### 2.7.4 Reading and Writing

`int ol_q_fscanf` (*FILE** `stream`, *double** `q`) [Function]
This routine reads formatted data from the stream *stream* into a quaternion $q$. $q$ must be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_q_fprintf` (*FILE** `stream`, *double** `q`, *char** `format`) [Function]
This routine writes a quaternion $q$ to the stream *stream*, using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.8 Pole Figure

A crystal orientation can be represented by a pole figure, in which specific *poles* of the crystal, e.g. {100}, are projected, see Figure 2.5.

One consider a reference sphere (with radius 1) attached to the reference coordinate system. The point of intersection of the pole with the reference sphere, situated in the upper hemisphere, is called $P$. Such a projection accounts for the crystal orientation – note that at least two poles are needed to describe unambiguously the orientation. The position of $P$ can be described by spherical polar coordinates, $(\alpha, \beta)$, see Figure 2.5 (left). $\alpha$ is the polar angle from the $Z_s$ axis with $\alpha \in [0, 90]$. $\beta$ is the azimuthal angle in the $X_s - Y_s$ plane from the $X_s$ axis with $\beta \in [0, 360[$. If $(hkl)$ is the pole of interest and $n = \sqrt{h^2 + k^2 + l^2}$, $\alpha$ and $\beta$ are given by,

$$\begin{pmatrix} \sin\alpha\cos\beta \\ \sin\alpha\sin\beta \\ \cos\alpha \end{pmatrix} = 1/n \begin{pmatrix} g_{11} & g_{21} & g_{31} \\ g_{12} & g_{22} & g_{32} \\ g_{13} & g_{23} & g_{33} \end{pmatrix} \begin{pmatrix} h \\ k \\ l \end{pmatrix}$$

Then, $P$ is projected onto the equatorial plane, $X_s - Y_s$, see Figure 2.5 (right). The position of the projection point, $p$, can be described by polar coordinates $(Op, \beta)$.

- The *stereographic projection* is commonly used in metallurgy since angular relationships in the crystal are preserved in the projection. $p$ is the point of intersection of segment $[PS]$ with the projection plane. $Op$ is given by,

$$Op = \tan(\alpha/2)$$

- The *equal-area projection* is commonly used in geology. Areas are preserved in the projection, so it is particularly appropriate for the measurements of population densities. $Op$ is given by,

$$Op = \sqrt{2}\sin(\alpha/2)$$

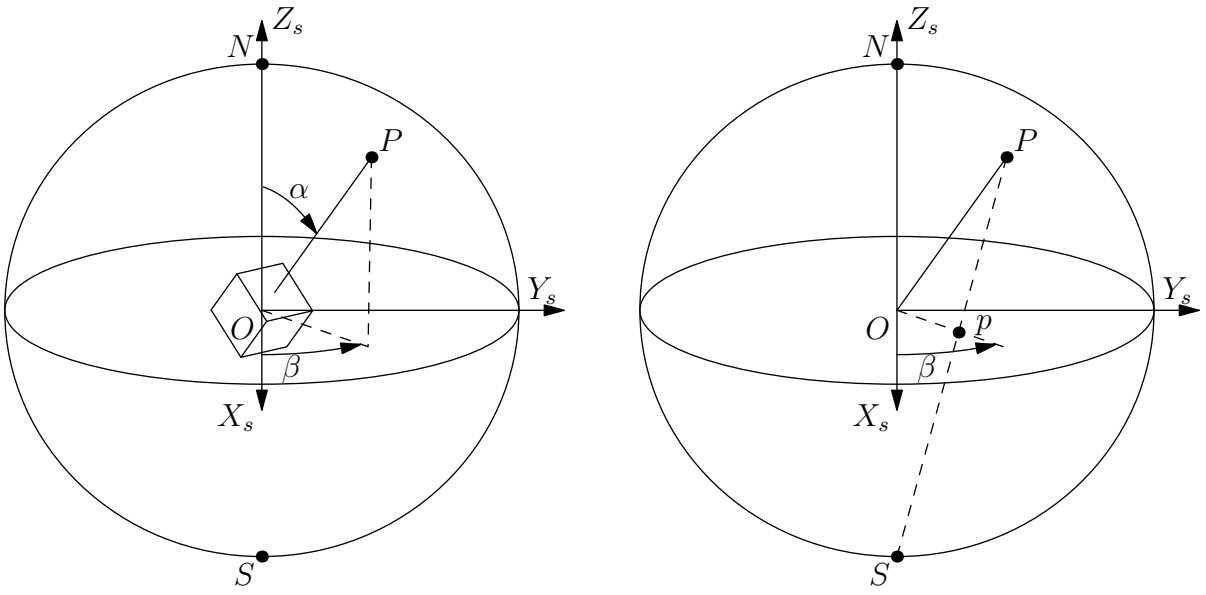



Figure 2.5: Construction of a pole figure. (left) Pole intersection with the reference sphere – case of the (100) pole. (right) Stereographic projection onto the equatorial plane.

### 2.8.1 Allocation

A *pole* is represented by a one-dimensional integer array, `pole[i]` with `i` in $\{0, 1, 2\}$; a vector *vect* is represented by a one-dimensional array, `pole[i]` with `i` in $\{0, 1, 2\}$; a projection point $p$ is represented by a one-dimensional array, `p[i]` with `i` in $\{0, 1\}$. For hexagonal lattices, a four indices pole can be represented by a one-dimensional array `hpole[i]` with `i` in $\{0, ..., 3\}$.

`int* ol_pole_alloc` (*void*) [Function]
`int* ol_hpole_alloc` (*void*) [Function]
`double* ol_vect_alloc` (*void*) [Function]
`double* ol_p_alloc` (*void*) [Function]
`void ol_pole_free` (*int* `pole`*) [Function]
`void ol_hpole_free` (*int* `hpole`*) [Function]
`void ol_vect_free` (*double* `vect`*) [Function]
`void ol_p_free` (*double* `p`*) [Function]
The four first routines create a pole, four indices pole, vector, projection point; they are initialized to 0. The four last routines free a previously allocated *pole*, four indices pole *hpole*, vector *vect* or projection point *p*.

### 2.8.2 Initializing and Copying

`void ol_pole_set_zero` (*int* `pole`*) [Function]
`void ol_vect_set_zero` (*double* `pole`*) [Function]
`void ol_p_set_zero` (*double* `pole`*) [Function]
These functions set a *pole*, vector *vect* or projection point $p$ to 0.

`void ol_pole_set_this` (*int* `pole`, *int* `polex`, *int* `poley`, *int* `polez`*) [Function]
`void ol_vect_set_this` (*double* `vect`, *double* `vectx`, *double* `vecty`, *double* `vectz`*) [Function]
`void ol_p_set_this` (*double* `p`, *double* `px`, *double* `py`*) [Function]
These functions set a *pole*, vector *vect* or projection point $p$ using the terms given in argument.

`void ol_pole_memcpy` (*int* `pole1`, *int* `pole2`*) [Function]
`void ol_vect_memcpy` (*double* `vect1`, *double* `vect2`*) [Function]
`void ol_p_memcpy` (*double* `p1`, *double* `p2`*) [Function]
This function copies a *pole*, vector *vect* or projection point $p$ into another. *pole2*, *vect2* or *p2* must be preallocated.

### 2.8.3 Pole to Unit Vector

`void ol_pole_vect` (*int* `pole`, *double* `vect`*) [Function]
This routine calculated the unit vector *vect* which corresponds to a pole *pole* (3 indices pole). *vect* must be preallocated.

`void ol_hpole_vect` (*int* `hpole`, *double* `vect`*) [Function]
This routine calculated the unit vector *vect* which corresponds to a pole *hpole* (4 indices pole, for hexagonal structure). *vect* must be preallocated.

### 2.8.4 Stereographic Projection

`void ol_g_stproj` (*double** `g`, *double* `vect`, *double* `p`*) [Function]
`void ol_g_stprojxy` (*double** `g`, *double* `vect`, *double* `p`*) [Function]
These routine calculated the stereographic projection $p$ of a pole whose corresponding unit vector is *vect*, for a crystal with rotation matrix $g$. The first routine returns polar coordinates ($p[0]$ is $Op$ and $p[1]$ is $\beta$). The second returns cartesian coordinates. $p$ must be preallocated.

### 2.8.5 Equal-area Projection

`void ol_g_eaproj` (*double** g, double* vect, double* p*) [Function]
`void ol_g_eaprojxy` (*double** g, double* vect, double* p*) [Function]

These routines calculated the equal-area projection $p$ of a pole whose corresponding unit vector is *vect*, for a crystal with rotation matrix $g$. The first routine returns polar coordinates ($p[0]$ is $Op$ and $p[1]$ is $\beta$). The second returns cartesian coordinates. $p$ must be preallocated.

### 2.8.6 Reading and Writing

`int ol_pole_fscanf` (*FILE* stream, int* pole*) [Function]
`int ol_hpole_fscanf` (*FILE* stream, int* hpole*) [Function]
`int ol_vect_fscanf` (*FILE* stream, double* vect*) [Function]
`int ol_p_fscanf` (*FILE* stream, double* p*) [Function]

These routines read formatted data from a stream *stream* into a *pole*, 4 indices pole *hpole*, vector *vect* or projection point $p$, respectively. *pole*, *vect* or $p$ must be preallocated. The functions return `0` for success and `EOF` if there was a problem reading from *stream*.

`int ol_pole_fprintf` (*FILE* stream, int* pole, char* format*) [Function]
`int ol_hpole_fprintf` (*FILE* stream, int* hpole, char* format*) [Function]
`int ol_vect_fprintf` (*FILE* stream, double* vect, char* format*) [Function]
`int ol_p_fprintf` (*FILE* stream, int* p, char* format*) [Function]

These routines write a *pole*, 4 indices pole *hpole*, vector *vect* or projection point $p$, respectively, to the stream *stream* using the format specifier *format*, which should be one of the `%d` formats for pole and `%g`, `%e` or `%f` formats for vector and projection point. The functions return a positive value for success and a negative value if there was a problem writing to *stream*.

## 2.9 Examples

### 2.9.1 Convert Euler Angles into Some Other Descriptors

This example (available from directory `examples/ex1`) illustrates how to convert Euler angles into rotation matrix, axis / angle of rotation, Rodrigues vector and quaternion.

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"

int
main (void)
{
  double*  e = ol_e_alloc ();
  double** g = ol_g_alloc ();
  double*  q = ol_q_alloc ();

  printf ("Euler angles?\n");
  ol_e_fscanf (stdin, e);

  ol_e_g (e, g);
  printf ("\nrotation matrix:\n");
  ol_g_fprintf (stdout, g, "%6.3f");

  printf ("\nquaternion:\n");
  ol_g_q (g, q);
  /* or directly ol_e_q (e, q); */

  ol_q_fprintf (stdout, q, "%.3f");

  ol_e_free (e);
  ol_g_free (g);
  ol_q_free (q);

  return EXIT_SUCCESS;
}
```

Here is what the example provides,

```
$ ex1
Euler angles?
30 0 0

rotation matrix:
 0.866  0.500  0.000
-0.500  0.866  0.000
 0.000 -0.000  1.000

quaternion:
0.966 0.000 0.000 0.259
```

### 2.9.2 Convert Euler Angles into Miller Indices

This example (available from directory `examples/ex2`) illustrates how to convert Euler angles into Miller indices, together with their qualities.

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"

int
main (void)
{
  int i, qty;
  double*  e    = ol_e_alloc ();
  double** g    = ol_g_alloc ();
  int**    allm = ut_alloc_2d_int (10, 6);
  double*  mq   = ut_alloc_1d (10);

  printf ("Euler angles?\n");
  ol_e_fscanf (stdin, e);

  ol_e_g (e, g);
  ol_g_m (g, 9, 5, allm, mq, &qty);

  printf ("\nMiller indices:\n");
  for (i = 0; i < qty; i++)
  {
    /* ol_m_mcubesym (allm[i], allm[i]); */
    printf ("(%4.2f)", mq[i]);
    ol_m_fprintf (stdout, allm[i], "%2d");
  }

  ol_e_free (e);
  ol_g_free (g);
  ut_free_2d_int (allm, 10);
  ut_free_1d (mq);

  return EXIT_SUCCESS;
}
```

Here is what the example provides,

```
$ ex2
Euler angles?
30 0 0

Miller indices:
(3.72) 0  0  1  2 -1  0
(4.85) 0  0  1  3 -2  0
(4.98) 0  0  1  5 -3  0
(6.13) 0  0  1  7 -4  0
(7.43) 0  0  1  4 -3  0
```

# 3 Orientation Generation

This chapter describes functions for generating orientations and misorientations. The input data are numbers between 0 and 1. Uniformly distributed inputs provide uniformly distributed outputs – this is why orientations and misorientations are distinguished: uniformly distributed orientations do not provide uniformly distributed misorientations, and vice-versa. Random data can be obtained through built-in functions that uses the ANSI C generator, but you may want to use your favorite generator instead.

Orientations can be generated either in the whole orientation space or in a smaller domain about the orientation space origin – to get them about another orientation, see Section 4.1.2 [Combination of Rotations], page 36. Misorientations are generated in a domain of angular value between 0 and 180 degrees.

The functions described in this chapter are available from files `ol_gen.[c,h]`. Due to dependencies, files `ol_des.[c,h]` and `ol_cal.[c,h]` must also be included.

## 3.1 Orientation Distribution

Here are functions to generate orientations. Uniformly distributed input numbers provide uniformly distributed orientations.

### 3.1.1 In the Whole Orientation Space

For orientations in the whole space, the usual way is by Euler angles. They are obtained from three numbers $n_1$, $n_2$ and $n_3$ between 0 and 1,

$$\begin{cases} \varphi_1 = 360\, n_1 \\ \phi \ = \operatorname{acos}(2\, n_2 - 1) \\ \varphi_2 = 360\, n_3 \end{cases}$$

`void ol_nb_e` (*double* `n1`, *double* `n2`, *double* `n3`, *double** `e`) [Function]
This routine computes Euler angles *e* from three numbers *n1*, *n2* and *n3* between 0 and 1.

`void ol_srand_e` (*int* `seed`, *int* `qty`, *double*** `e`) [Function]
This routine generates random orientations expressed as Euler angles. *seed* is the pseudo-random generator seed, *qty* is the quantity of orientations and *e[i]* with $i \in \{0, ..., qty - 1\}$ are the generated orientations.

### 3.1.2 Orientation Spread

Spreads of orientation are obtained through the axis/angle of rotation descriptor, $(r, \theta)$. It is obtained from three numbers $n_1$, $n_2$ and $n_3$ between 0 and 1, and a maximum angular value $\theta_{max}$. The axis $r$ is obtained from the two first numbers, from which spherical polar coordinates $\alpha, \beta$ are calculated. See Figure 2.5 for an illustration of how the angles are defined – replace $[OP]$ by $r$. Here, the angles are defined in the general ranges: $\alpha \in [0, 180]$ and $\beta \in [0, 360[$. They are obtained as follows,

$$\begin{cases} \alpha = \operatorname{acos}(2\, n_1 - 1) \\ \beta = 360 n_2 \end{cases}$$

By means of small approximations, $\theta$ can be obtained from $n_3$ and $\theta_{max}$, as follows,

$$\theta = \theta_{max} {n_3}^{\frac{1}{3}}$$

`void ol_nb_r` (*double* `n1`, *double* `n2`, *double** `r`) [Function]
This routine computes an axis of rotation *r* from two numbers *n1* and *n2* between 0 and 1.

`void ol_nb_max_theta` (*double* `n3`, *double* `thetamax`, *double** `&theta`) [Function]
This routine computes an angle of rotation *theta* lower than *thetamax* from number *n3* between 0 and 1.

Here is a function to get the axis/angle of rotation (*r*, *theta*) directly,

`void ol_nb_max_rtheta` (*double* `n1`, *double* `n2`, *double* `n3`, *double* `thetamax`, *double** `r`, *double** `&theta`) [Function]
This routine computes an axis/angle of rotation (*r*, *theta*) with *theta* lower than *thetamax* from three numbers *n1*, *n2* and *n3* between 0 and 1.

## 3.2 Misorientation Distribution

Here are functions to generate misorientations. Uniformly distributed input numbers provide uniformly distributed misorientations. The axis / angle of rotation representation is used. Misorientation axes can be generated as for orientations, see Section 3.1.2 [Ori. Distrib. Axis/Angle of Rotation], page 31, but not misorientation angles.

Here, $\theta$ is obtained from $n_3$, as follows,

$$\theta = \theta_{max} n_3$$

`void ol_nb_max_theta_mis` (*double* `n3`, *double* `thetamax`, *double** `&theta`) [Function]
This routine computes an angle of rotation *theta* from number *n3* between 0 and 1. It provides an angle in $[0, \theta_{max}]$.

Here is a function to get the misorientation axis/angle of rotation (*r*, *theta*) directly,

`void ol_nb_max_rtheta_mis` (*double* `n1`, *double* `n2`, *double* `n3`, *double* `thetamax`, *double** `&theta`) [Function]
This routine computes an angle of rotation *theta* from number *n3* between 0 and 1. It provides an angle in $[0, \theta_{max}]$.

## 3.3 Examples

Here are two examples of orientation generation. The input numbers are obtained thanks to the (poor quality) `rand()` ANSI C generator.

### 3.3.1 Random Orientation Generation

This example (available from directory `examples/ex3`) illustrates how to generate 500 random orientations under the form of Euler angles. They are written to the prompt. The orientations are plotted on Figure 3.1 (left).

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"
#define frand() ((double) rand() / (RAND_MAX+1.0))

int
main (void)
{
  int i;
  double * e = ol_e_alloc ();
  srand (1);
```

```
  for (i = 0; i < 500; i++)
    {
      ol_nb_e (frand (), frand (), frand (), e);
      ol_e_fprintf (stdout, e, "%16.12f");
    }

  ol_e_free (e);

  return EXIT_SUCCESS;
}
```

### 3.3.2 Generation of a Spread of Orientations

This example (available from directory `examples/ex4`) illustrates how to generate a spread of 200 orientations, with 15 degrees maximum disorientation. The orientations are written to the prompt. They are plotted on Figure 3.1 (right).

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"
#define frand() ((double) rand() / (RAND_MAX+1.0))

int
main (void)
{
  int i;
  double theta;
  double **g = ol_g_alloc ();
  double *e = ol_e_alloc ();
  double *r = ol_r_alloc ();
  srand (3);

  for (i = 0; i < 200; i++)
  {
    ol_nb_max_rtheta (frand (), frand (), frand (), 15, r, &theta);
    ol_rtheta_g (r, theta, g);
    ol_g_e (g, e);
    ol_e_fprintf (stdout, e, "%16.12f");
  }

  ol_e_free (e);
  ol_g_free (g);
  ol_r_free (r);

  return EXIT_SUCCESS;
}
```

### 3.3.3 Generation of Misorientations

The generation of misorientations is very similar to that of orientation spreads. Just change `ol_nb_max_rtheta` to `ol_nb_max_rtheta_mis` is the previous example. The misorientations are plotted on Figure 3.2.

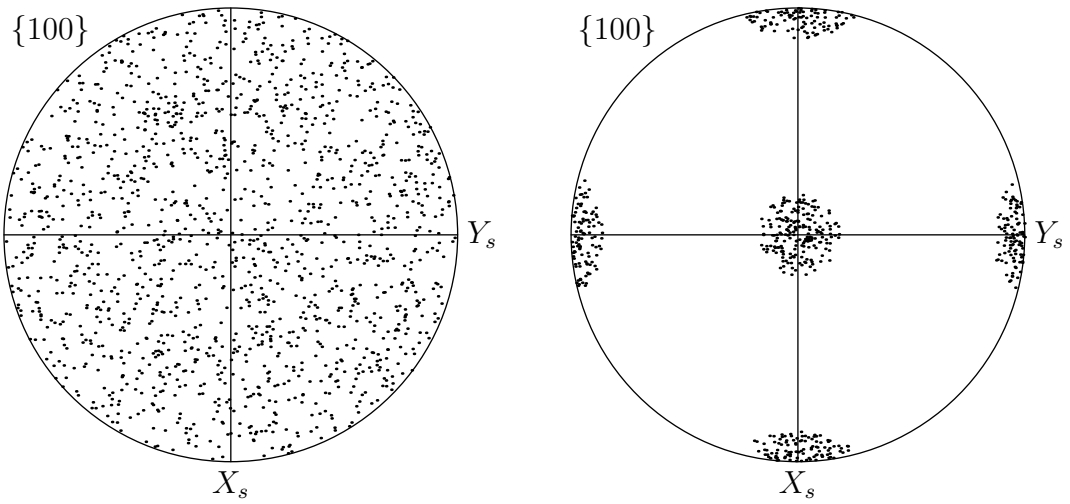



Figure 3.1: Generated orientations (equal-area projection). (left) In the whole orientation space, (right) with a maximum misorientation of 15 degrees about the orientation space origin.

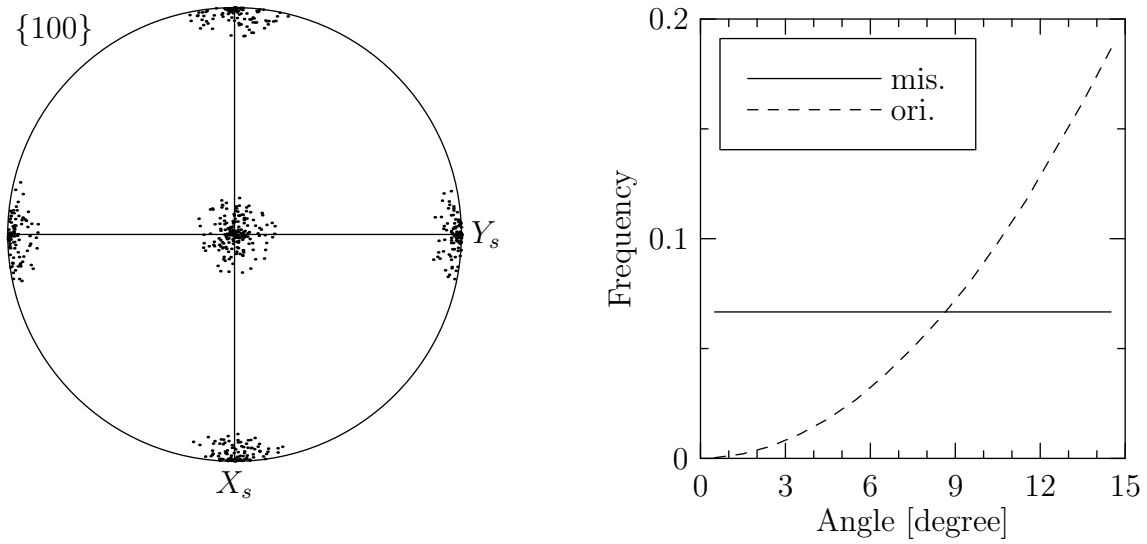



Figure 3.2: Generated misorientations with a maximum value of 15 degrees. (left) equal-area projection that illustrates that uniform misorientations do not show uniform "orientation distribution". (right) Distribution of the misorientation angle, which is uniform for uniformly distributed misorientations, but not for uniformly distributed orientations.

# 4 Orientation Calculation

This chapter describes functions for performing calculations with orientations which range from low-level operations: inverse rotation or combination of rotations, to high-level operations: disorientation calculations, orientation averaging, etc.

From a general point of view, the functions are available for rotation matrix and quaternion, but it may not be the case for functions which requires a specific descriptor, such as the orientation averaging function which can be done from quaternions only. Furthermore, as rotations/orientations can be expressed in different coordinate systems, sometimes several versions of the functions are available.

The functions described in this chapter are available from files `ol_cal.[c,h]`. Due to dependencies, file `ol_des.[c,h]` must also be included.

## 4.1 Elementary Operations

Here are described the elementary operations on which higher-level calculations are based.

### 4.1.1 Inverse Rotation

The calculation of an inverse rotation is self-explanatory (see Figure 4.1). The properties of the rotation matrix (orthogonal) and quaternion (unitary) provide additional relations,

$$g_{inv} = g^{-1} = g^t$$
$$q_{inv} = q^{-1} = q^c$$

where $g^t$ is the transpose of $g$ and $q^c$ is the conjugate of $q$.

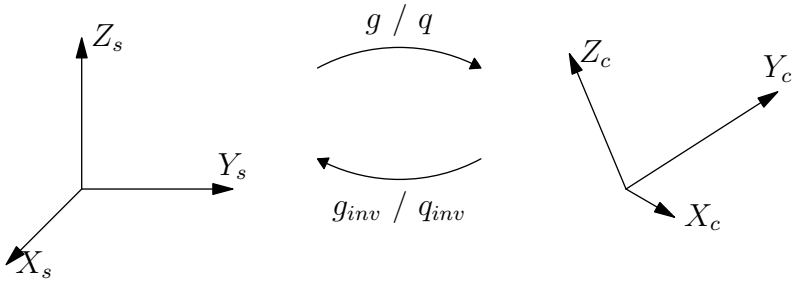



Figure 4.1: Inverse rotation.

`void ol_g_inverse` (*double** g, double** ginv*) [Function]
`void ol_q_inverse` (*double* q, double* qinv*) [Function]

These routines compute the inverse, *ginv* / *qinv*, of $g$ / $q$.

### 4.1.2 Combination of Rotations

The rotation $g_3$ / $q_3$ resulting from two successive rotations, $g_1$ / $q_1$ then $g_2$ / $q_2$, see Figure 4.2, expressed in their *current coordinate systems*: rotation 1 in $C_0$ and rotation 2 in $C_1$, is given by,

$$g_3 = g_2 g_1$$
$$q_3 = q_1 q_2$$

When both $g_1$ / $q_1$ and $g_2$ / $q_2$ are expressed in the *same, reference coordinate system* $C_0$, $g_3$ / $q_3$ is given by,

$$g_3 = g_1 g_2$$
$$q_3 = q_2 q_1$$

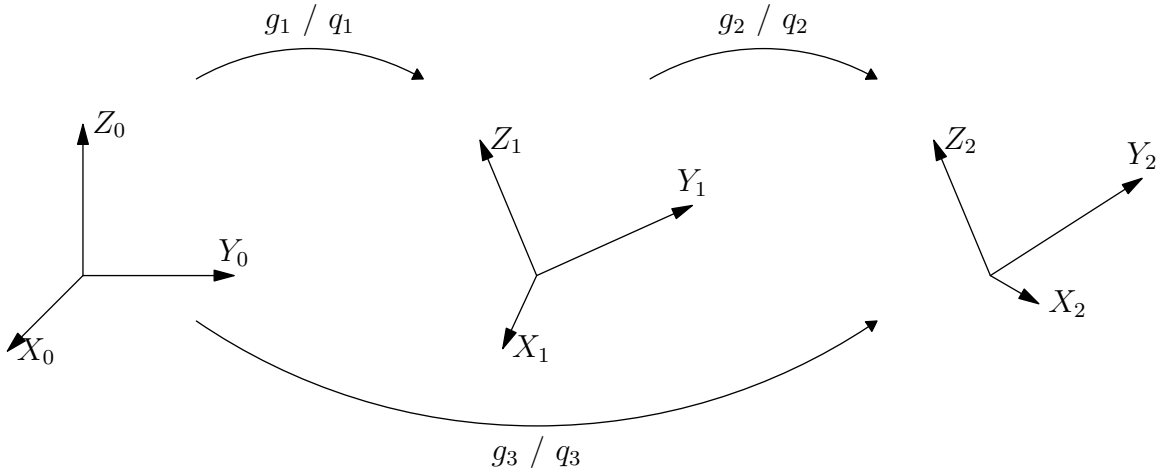



Figure 4.2: Combination of rotations.

`void ol_g_g_g_cur` (*double** `g1`, *double** `g2`, *double** `g3`) [Function]
`void ol_q_q_q_cur` (*double** `q1`, *double** `q2`, *double** `q3`) [Function]
`void ol_g_g_g` (...) [*same function, shorter name*] [Function]
`void ol_q_q_q` (...) [*same function, shorter name*] [Function]

These routines compute the rotation, *g3 / q3*, resulting from two successive rotations expressed in their current coordinate systems, *g1 / q1* then *g2 / q2*.

`void ol_g_g_g_ref` (*double** `g1`, *double** `g2`, *double** `g3`) [Function]
`void ol_q_q_q_ref` (*double** `q1`, *double** `q2`, *double** `q3`) [Function]

These routines compute the rotation, *g3 / q3*, resulting from two successive rotations expressed in the same, reference coordinate system, *g1 / q1* then *g2 / q2*.

## 4.2 Change in Coordinate System

It is sometimes needed to express an orientation in a different coordinate system, see Figure 4.3. Let $C_s$ and $C_s{}'$ be the old and new coordinate system, respectively. $g_s$ / $q_s$ is the rotation that brings the old coordinate system to the new coordinate system. Let $g$ / $q$ be the orientation of a crystal expressed in the old coordinate system. Its expression in the new coordinate system, $g'$ / $q'$, is given by,

$$g' = g{g_s}^{-1}$$
$$q' = {q_s}^{-1}q$$

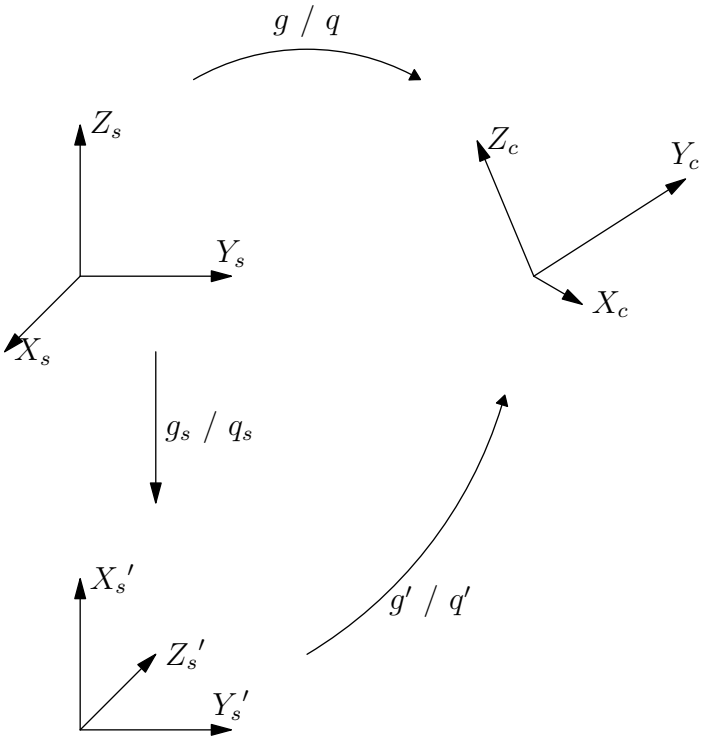



Figure 4.3: Change in coordinate system. $C_s$ and $C_s{}'$ are the old and new coordinate system. $C_c$ is the crystal coordinate system.

`void ol_g_csys` (*double** g, double** gs, double** g'*) [Function]
`void ol_q_csys` (*double* q, double* qs, double* q'*) [Function]

These routines express an orientation in a new coordinate system. *g* / *q* is the orientation in the old coordinate system. *gs* / *qs* is the rotation that brings the old coordinate system to the new coordinate system. *g'* / *q'* is the orientation in the new coordinate system.

## 4.3 Crystallographically-related Orientations

Due to crystal symmetry, there are several ways in which the crystal can be arranged, see Figure 4.4. Each one is described by a particular orientation descriptor ($g$, $q$, $m$, ...).

As for the rotation matrix description $g$, the crystallographically-related solutions are obtained by multiplying the rotation matrix $g$ by a symmetry operator $T_i$. Similarly, as for the quaternion $q$, they are obtained by multiplying the quaternion $q$ by a symmetry operator $U_i$,

$$g_i = T_i g$$
$$q_i = q U_i$$

As for the Miller indices $m$, they are obtained by permuting the indices.

For cubic crystals, there are 24 matrices $T_i$, quaternions $U_i$, or Miller indices permutations $p_i$: the identity, the three rotations of 90 degrees about each of the three $\langle 100 \rangle$, one rotation of 180 degrees about each of the six $\langle 110 \rangle$ and the two rotations of 120 degrees about each of the four $\langle 111 \rangle$. For the sake of conciseness, the matrices, quaternions and Miller indices permutations are given in Appendix B [Cubic Symmetry Operators], page 63.

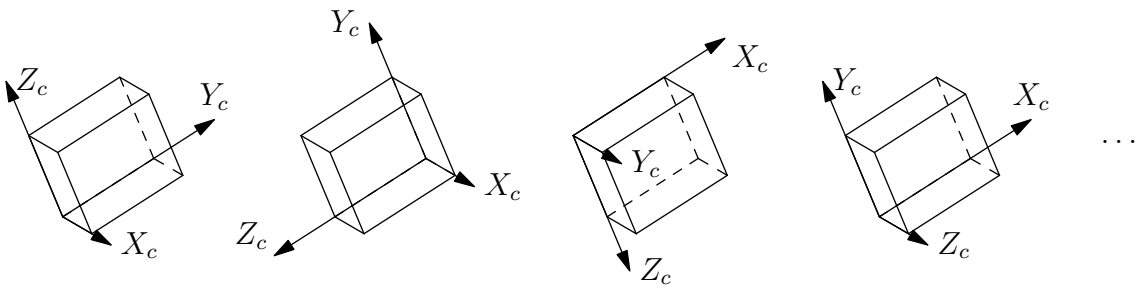



Figure 4.4: Some crystallographically-related orientations in the case of cubic symmetry.

`void ol_g_cubesym` (*double** `g`, *int* `i`, *double** `g2`) [Function]
`void ol_q_cubesym` (*double** `q`, *int* `i`, *double** `q2`) [Function]
`void ol_m_cubesym` (*int** `m`, *int* `i`, *int** `m2`) [Function]

These routines compute the $i^{th}$ crystallographically-related orientation ($i \in \{1, ..., 24\}$), *g2* / *q2* / *m2*, of *g* / *q* / *m*.

`double ol_g_gcubesym` (*double** `g`, *double** `g2`) [Function]
`double ol_q_qcubesym` (*double** `q`, *double** `q2`) [Function]
`void ol_m_mcubesym` (*int** `m`, *int** `m2`) [Function]
`void ol_e_ecubesym` (*double** `e`, *double** `e2`) [Function]

The two first routines compute the preferential rotation matrix *g2* or quaternion *q2* among the 24 crystallographically-related orientations of *g* / *q* ; it is the one whose angle of rotation is the smallest. The angle is returned.

The third routine computes the preferential Miller indices *m2* among the 24 crystallographically-related orientations of *m*; it is the one which matches the following criteria (listed in descending priority): *(i)* $(hkl)$ with positive values, *(ii)* $[uvw]$ with as less negative values as possible, *(iii)* $(hkl)$ with increasing values, *(iv)* $[uvw]$ with descending absolute values, *(v)* $[uvw]$ with negative values as at right as possible. This yields a unique solution.

The fourth routine computes the preferential Euler angles *e2* among the 24 crystallographically-related orientations of *e*; it is the one with $(\varphi_1 \in [0, 180], \phi \in [0, 90], \varphi_2 \in [0, 90])$.[1]

[1] Actually, there can be more that one solution in this domain – the one with the smallest $\varphi_1$ is chosen.

## 4.4 Symmetry with respect to Reference Coordinate System Planes

Usually the specimen shows one or more plane(s) of symmetry. For example, at the middle of a rolled sheet, there are two planes of symmetry, whose normals are $X_s$ and $Y_s$, respectively (*orthotropic* symmetry). In such cases, there are several ways in which the reference coordinate system can be arranged w.r.t. the specimen. It is equivalent to consider a unique reference coordinate system and that there are several ways in which the crystal coordinate system can be arranged w.r.t. that coordinate system.

Figure 4.5 is an example of symmetry w.r.t. the plane of normal $Y_s$. The symmetrical solution for the crystal coordinate system is obtained in two steps. The first one is to apply symmetry, changing the basis vector $y$-coordinates to their opposites. At this point, the new coordinate system is left-handled. The second step is to make it right-handled by changing the $y$-basis vector to its opposite.

Let planes $X_s$, $Y_s$ and $Z_s$ be noted $P_i$, with $i = 1, 2, 3$ respectively, and let $P_i$ be the plane of interest. In mathematical terms, as for the rotation matrix description $g$, the solution for a symmetry w.r.t. $P_i$, called $g_i$, is obtained as follows,

$$g_i = m_i \left(g {m_i}^{-1}\right)$$

where $m_i$ is a $3 \times 3$ symmetry matrix given by,

$$\begin{aligned} \left(m_i\right)_{jj} &= \begin{cases} 1 \text{ for } j \neq i \\ -1 \text{ for } j = i \end{cases} \\ \left(m_i\right)_{jk} &= 0 \end{aligned}$$

As for the quaternion description $q$, the solution $q_i$ is obtained as follows,

$$q_0 = \rho,\ q_1 = \lambda,\ q_2 = \mu,\ q_3 = \nu$$

$$\left(q_i\right)_j = \begin{cases} q_j \text{ if } j = i \\ -q_j \text{ if } j \neq i \end{cases}$$

As for the Miller indices description $m = (hkl)\,[uvw]$, the solution $m_i$ is obtained as follows,

$$\begin{aligned} m_1 &= (\overline{h}kl)\,[u\overline{v}\overline{w}] \\ m_2 &= (h\overline{k}l)\,[u\overline{v}w] \\ m_3 &= (\overline{h}\overline{k}l)\,[uv\overline{w}] \end{aligned}$$

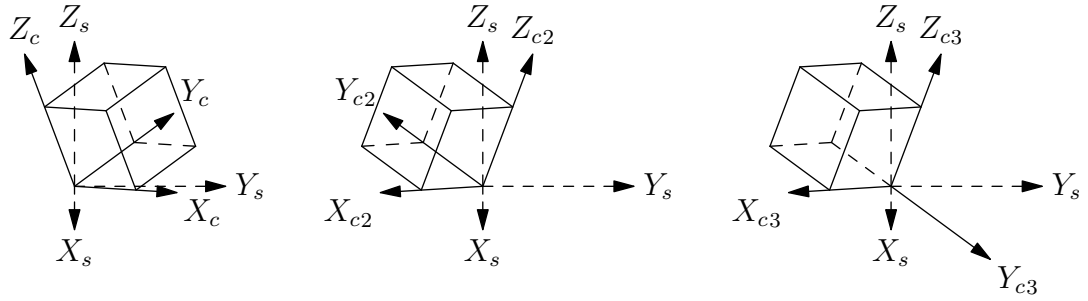



Figure 4.5: Symmetry w.r.t. a reference coordinate system plane.

`void ol_g_refsym` (*double** `g`, *int* `i`, *double** `g2`) [Function]
`void ol_q_refsym` (*double** `q`, *int* `i`, *double** `q2`) [Function]
`void ol_m_refsym` (*int** `m`, *int* `i`, *int** `m2`) [Function]
These routines compute the symmetrical orientation w.r.t. plane $i$ ($i \in \{1, 2, 3\}$), *g2* / *q2* / *m2* of *g* / *q* / *m*. *g2* / *q2* / *m2* must be preallocated.

`void ol_g_refsym_monoclinic` (*double** `g`, *double*** `g2`, *int** `&qty`) [Function]
`void ol_q_refsym_monoclinic` (*double** `q`, *double** `q2`, *int** `&qty`) [Function]
`void ol_m_refsym_monoclinic` (*int** `m`, *int** `m2`, *int** `&qty`) [Function]
These routines compute the set of symmetrical orientations of *g* / *q* / *m* in the case of monoclinic symmetry (symmetry w.r.t. plane 2). The results are stored in *g2* / *q2* / *m2*. In the general case, 2 different orientations are returned, but in the case of symmetrical orientations, they can be less solutions (down to 1). *g2* / *q2* / *m2* must be preallocated.

`void ol_g_refsym_orthorhombic` (*double** `g`, *double*** `g2`, *int** `&qty`) [Function]
`void ol_q_refsym_orthorhombic` (*double** `q`, *double** `q2`, *int** `&qty`) [Function]
`void ol_m_refsym_orthorhombic` (*int** `m`, *int** `m2`, *int** `&qty`) [Function]
These routines compute the set of symmetrical orientations of *g* / *q* / *m* in the case of orthorhombic symmetry (symmetry w.r.t. planes 1 and 2). The results are stored in *g2* / *q2* / *m2*. In the general case, 4 different orientations are returned, but in the case of symmetrical orientations, they can be less solutions (down to 1). *g2* / *q2* / *m2* must be preallocated.

`void ol_g_refsym_g` (*double** `g`, *int* `i`, *double** `gs`) [Function]
`void ol_q_refsym_q` (*double** `q`, *int* `i`, *double** `qs`) [Function]
`void ol_m_refsym_g` (*int** `m`, *int* `i`, *double** `gs`) [Function]
These routines compute the misorientation between orientation *g* / *q* / *m* and plane $i$ ($i \in \{1, 2, 3\}$), recorded in *gs* / *qs* / *gs*, taken as half the misorientation between the orientation and its symmetrical w.r.t. plane *i*.

## 4.5 Misorientations, Disorientations

Let $g_1$ / $q_1$ and $g_2$ / $q_2$ be two orientations, see Figure 4.6. The *misorientation* between the two orientations is defined as the rotation $g_m$ / $q_m$ that brings one orientation to the other, say, orientation 1 to orientation 2.

The misorientation is commonly expressed in the *reference coordinate system*. In this case, it is given by,

$$\begin{aligned} g_m &= {g_1}^{-1} g_2 \\ q_m &= q_2 {q_1}^{-1} \end{aligned}$$

When expressed in the *coordinate system of crystal 1*, the misorientation is given by,

$$\begin{aligned} g_m &= g_2 {g_1}^{-1} \\ q_m &= {q_1}^{-1} q_2 \end{aligned}$$

Actually, due to crystal symmetry, there are several misorientations, $g_{mi}$ with $i \in \{1, ..., n\}$, which are obtained by replacing orientation 2 ($g_2$ / $q_2$) by its $n$ crystallographically-related orientations, see Section 4.3 [Crystallographically-related Orientations], page 38. The *disorientation*, $g_d$ / $q_d$, is defined as the misorientation with the minimum angle of rotation.

In mathematical terms, when expressed in the *reference coordinate system*, it is given by,

$$\begin{cases} g_{mi} = {g_1}^{-1} \mathrm{sym}_i(g_2) \\ g_d = g_{mj} \text{ where } j \,|\, \forall\, i \in \{1, ..., n\}\,,\ \theta(g_{mj}) \leq \theta(g_{mi}) \end{cases}$$

$$\begin{cases} q_{mi} = \mathrm{sym}_i(q_2) {q_1}^{-1} \\ q_d = q_{mj} \text{ where } j \,|\, \forall\, i \in \{1, ..., n\}\,,\ \theta(q_{mj}) \leq \theta(q_{mi}) \end{cases}$$

where $\mathrm{sym}_i(\bullet)$ is the $i^{th}$ crystallographically-related orientation and $\theta(\bullet)$ is the rotation amplitude associated to entity $\bullet$ (see Section 2.5 [Axis/Angle of Rotation], page 18).

When expressed in the *coordinate system of crystal 1*, it is given by,

$$\begin{cases} g_{mi} = \mathrm{sym}_i(g_2) {g_1}^{-1} \\ g_d = g_{mj} \text{ where } j \,|\, \forall\, i \in \{1, ..., n\}\,,\ \theta(g_{mj}) \leq \theta(g_{mi}) \end{cases}$$

$$\begin{cases} q_{mi} = {q_1}^{-1} \mathrm{sym}_i(q_2) \\ q_d = q_{mj} \text{ where } j \,|\, \forall\, i \in \{1, ..., n\}\,,\ \theta(q_{mj}) \leq \theta(q_{mi}) \end{cases}$$

Moreover, for cubic crystal symmetry,

$$\begin{aligned} \theta(g_{mi}) \leq 45 &\Longrightarrow g_{mj} = g_{mi} \\ \theta(q_{mi}) \leq 45 &\Longrightarrow q_{mj} = q_{mi} \end{aligned}$$

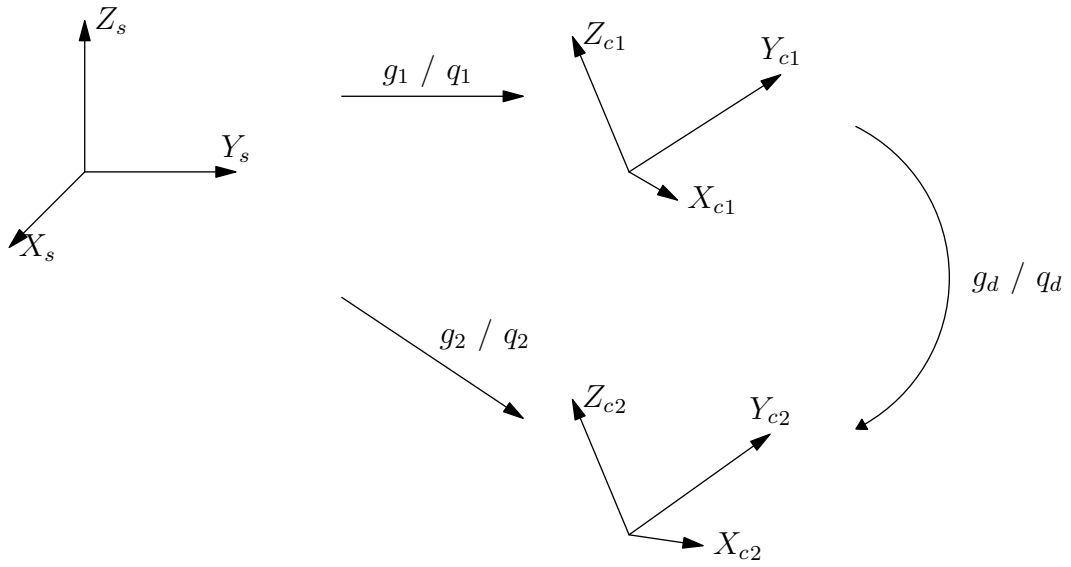



Figure 4.6: Disorientation $g_d$ / $q_d$ between two crystals of orientations $g_1$ / $q_1$ and $g_2$ / $q_2$.

`int ol_g_g_gdisori_ref` (*double** `g1`, *double** `g2`, *double** `gd`) [Function]
`int ol_q_q_qdisori_ref` (*double** `q1`, *double** `q2`, *double** `qd`) [Function]
`int ol_g_g_gdisori` (...) [*same function, shorter name*] [Function]
`int ol_q_q_qdisori` (...) [*same function, shorter name*] [Function]
This routine computes the disorientation *gd* / *qd* between two orientations *g1* / *q1* and *g2* / *q2*. Cubic crystal symmetry is assumed. The disorientation is expressed in the reference coordinate system. *d* is returned.

`int ol_g_g_gdisori_cur` (*double** `g1`, *double** `g2`, *double** `gd`) [Function]
`int ol_q_q_qdisori_cur` (*double** `q1`, *double** `q2`, *double** `qd`) [Function]
This routine computes the disorientation *gd* / *qd* between two orientations *g1* / *q1* and *g2* / *q2*. Cubic crystal symmetry is assumed. The disorientation is expressed in the coordinate system of crystal 1. *d* is returned.

`int ol_g_g_disori` (*double** `g1`, *double** `g2`, *double* `&theta`) [Function]
`int ol_q_q_disori` (*double** `q1`, *double** `q2`, *double* `&theta`) [Function]
These fast routines return the disorientation angle *theta* between two orientations *q1* and *q2*. Cubic crystal symmetry is assumed. (*theta* does not depend upon the coordinate system in which it is calculated.) *d* is returned.

An additional "shortcut" function is available:

`int ol_e_e_disori` (*double** `e1`, *double** `e2`, *double* `&theta`) [Function]
This routine returns the disorientation angle *theta* between two orientations expressed as Euler angles, *e1* and *e2*. Cubic crystal symmetry is assumed. *d* is returned.

**Note:** To get *mis*orientations, replace `disori` by `misori` is the function names, above.

**Caution:** Functions `ol_g_g_gdisori ()` and `ol_q_q_qdisori ()` are shortcuts for `ol_g_g_gdisori_ref ()` and `ol_q_q_qdisori_ref ()`, contrary to functions `ol_g_g_g ()` and `ol_q_q_q ()` which are shortcuts for `ol_g_g_g_cur ()` and `ol_q_q_q_cur ()`.

Here are additional functions to compute the misorientation between two vectors.

`void ol_vect_vect_theta` (*double* v1*, *double* v2*, *double* `&theta`) [Function]
`void ol_vect_vect_rtheta` (*double* v1*, *double* v2*, *double* `r`*, *double* `&theta`) [Function]

This routine computes the angle of rotation *theta* between two vectors *v1* and *v2*.

## 4.6 Examples

### 4.6.1 Disorientations – Mackenzie Distribution

This example (available from directory `examples/ex5`) illustrates how to compute disorientations from the reference coordinate system. Their distribution is recorded. The orientations are generated randomly thanks to the (poor quality) `rand()` ANSI C generator. As can be seen on Figure 4.7, the disorientations follow the well-known Mackenzie distribution.

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"
#define frand() ((double) rand() / (RAND_MAX+1.0))

int
main (void)
{
  int i;
  FILE *out;
  double *e = ol_e_alloc ();
  double *q  = ol_q_alloc ();
  double dis;
  int * qty = ut_alloc_1d_int (63);

  for (i = 0; i < 1000000; i++)
    {
      ol_nb_e (frand (), frand (), frand (), e);
      ol_e_q (e, q);
      dis = ol_q_qcubesym (q, q);
      qty [(int)dis]++;
    }

  out = fopen ("ex5.out", "w");
  for (i = 0; i < 63; i++)
    printf ("%2d %2d %f\n", i, i+1, (double)qty[i]/1000000);
  fclose (out);

  ol_e_free (e);
  ol_q_free (q);

  return EXIT_SUCCESS;
}
```

### 4.6.2 Symmetry w.r.t. the reference coordinate system

This example (available from directory `examples/ex11`) illustrates how to compute symmetrical (orthorhombic) orientations. The result is illustrated on Figure 4.8.

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"
#define frand() ((double) rand() / (RAND_MAX+1.0))

int
main (void)
{
  int i, qty;
  FILE* file;
  double * e = ol_e_alloc ();
  double * q = ol_q_alloc ();
  double ** qsym = ut_alloc_2d (4, 4);

  file = fopen ("ex11.in", "r");
  ol_e_fscanf (file, e);
  fclose (file);

  ol_e_q (e, q);
  ol_q_refsym_orthorhombic (q, qsym, &qty);

  file = fopen ("ex11.out", "w");
  for (i = 0; i < qty; i++)
  {
    ol_q_e (qsym[i], e);
    ol_e_fprintf (file, e, "%6.1f");
  }
  fclose (file);

  return EXIT_SUCCESS;
}
```

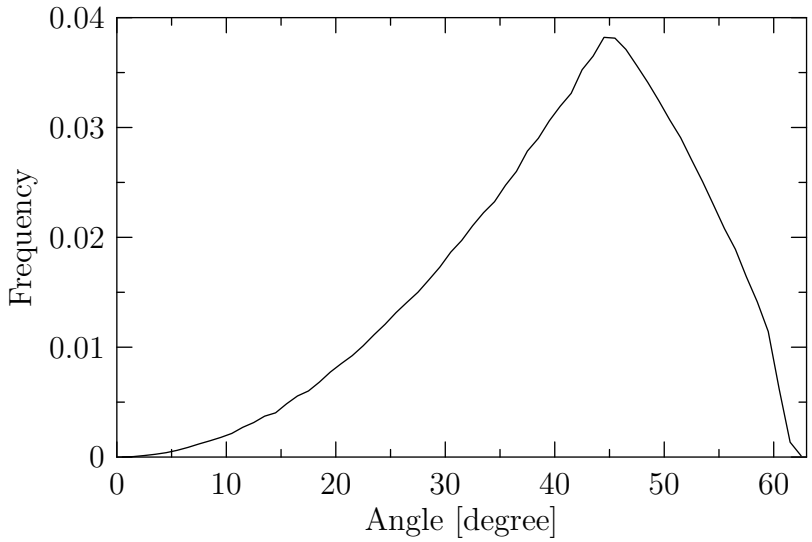



Figure 4.7: The Mackenzie distribution.

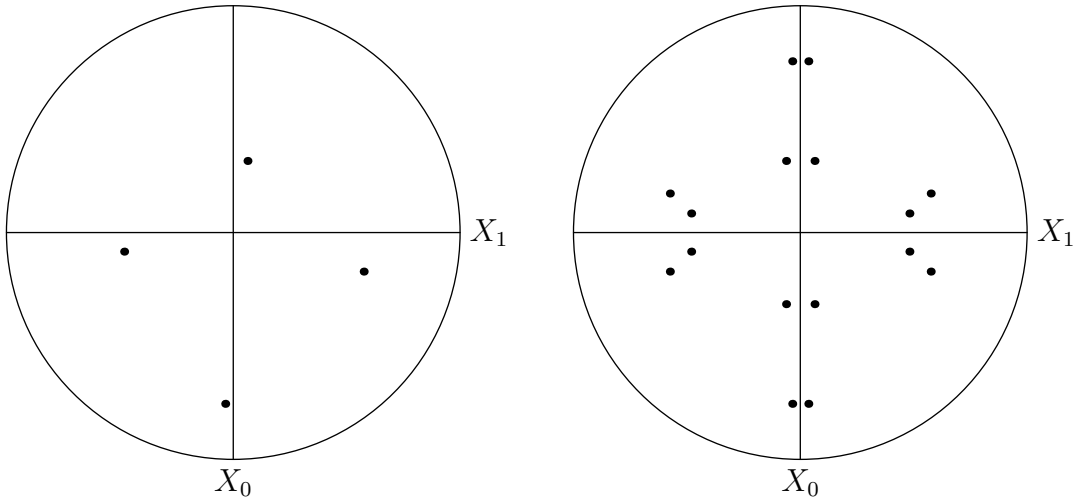



Figure 4.8: Symmetry w.r.t. the reference coordinate system. (left) Original orientation, (right) set of symmetrical orientations in the case of orthorhombic symmetry.

# 5 Orientation Set

This chapter describes functions for treating orientation sets, for example orientation averaging or characterization of spreads.

In the library, an orientation set is defined by an `OL_SET` structure. It contains only two components: the number of orientations and the orientations recorded as quaternions.

```
typedef struct
{
  size_t   size;
  double** q;
} OL_SET;
```

The functions described in this chapter are available from files `ol_set.[c,h]`. Due to dependencies, files `ol_des.[c,h]` and `ol_cal.[c,h]` must also be included.

## 5.1 Allocation

An orientation set is represented by an `OL_SET` structure.

`struct OL_SET ol_set_alloc` (*int size*) [Function]
`void ol_set_free` (*struct OL_SET set*) [Function]
The first routine creates a set of *size* orientations. The second frees a previously allocated orientation *set*.

## 5.2 Mean Orientation

An expression for the mean orientation has been proposed by (Humphreys *et al.*, 2001) and more generally by (Glez and Driver, 2001), using the quaternions. The main expression for the mean orientation $q_{mean}$ of the set of orientations $q_i$ with $i \in \{1, 2, ..., n\}$ is (see Figure 5.1),

$$q_{mean} = \left( \sum_{i=1}^{n} q_i \right) / \left( |\sum_{i=1}^{n} q_i| \right)$$

which is so that,

$$\left( \sum_{i=1}^{n} \theta_d(q_{mean}, q_i)^2 \right) \text{ minimum}$$

where $\theta_d(\bullet_1, \bullet_2)$ is the disorientation angle between $\bullet_1$ and $\bullet_2$.

Actually, the calculation of the mean orientation requires particular precautions since,

- an orientation can be expressed by different quaternions, due to crystal symmetry and equivalency of $q$ and $-q$ ($24 \times 2$ possibilities for cubic crystals).

  The solution would be to consider the crystallographically-related orientations which are the less disoriented from, e.g., the reference coordinate system, and to take them positive.

- the set of quaternions can be submitted to the so-called *umklapp effect*.

  The solution is to express the quaternions not w.r.t. the reference coordinate system, but w.r.t. an orientation that "would not be too far from the mean orientation". For small orientation spreads, anyone of the orientations can be used (e.g. the first one).

So, the actual expression of $q_{mean}$ is,

$$q_{mean} = q_{ref} \left( \sum_{i=1}^{n} q_d(q_{ref}, q_i) \right) / \left( |\sum_{i=1}^{n} q_d(q_{ref}, q_i)| \right)$$

where $q_{ref}$ is the reference orientation and $q_d(\bullet_1, \bullet_2)$ is the disorientation quaternion between $\bullet_1$ and $\bullet_2$ expressed in the coordinate system of $\bullet_1$.

Large orientation spread can be submitted to the umklapp effect depending on the reference orientation, so it is better to use an iterative procedure where, at each iteration, the reference orientation used is the previously calculated mean orientation.

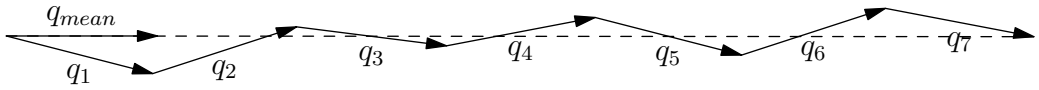



Figure 5.1: Calculation of a mean orientation. $q_1, ..., q_7$ are the orientations to average; $q_{mean}$ is the mean orientation.

`void ol_set_mean` (*struct OL_SET* `set`, *double** `qref`, *double** `qmean`) [Function]
This routine computes the mean orientation *qmean* of the orientation *set*, using *qref* as reference. Cubic crystal symmetry is assumed. The first orientation is taken as the reference orientation.

`int ol_set_mean_iter` (*struct OL_SET* `set`, *double** `qref`, *double** `qmean`) [Function]
This routine is similar to `ol_set_mean`, but applies an iterative procedure which is required for large orientation spreads. The returned value is `0` in the case of convergence, `-1` else.

## 5.3 Orientation Spread Anisotropy

The characterization of an anisotropic spread of orientations has been described in (Glez and Driver, 2001). The spread has to be centred in the orientation space (i.e., mean = origin).[1] Such a characterization is not a straightforward operation, but by means of small approximations, it can be treated in the Rodrigues space (see Section 2.6 [Rodrigues Vector], page 21), which is three-dimensional and Euclidean.

An anisotropic spread of vectorial data is described by a covariant matrix $S$ (of components $S_{ij}$). If the $n$ orientations are described by Rodrigues vectors $R$, the matrix $S$ is defined by,

$$S_{ij} = E\left(R_i R_j\right) - E\left(R_i\right) E\left(R_j\right)$$

where $E(\bullet)$ is the average of $\bullet$ on the $n$ orientations,

$$E(\bullet) = 1/n \sum_{k=1}^{n} \bullet_k$$

$S$ is a $3 \times 3$ symmetric matrix and can be diagonalized. The eigenvectors $v_i$ with $i \in \{0, 1, 2\}$, define the principal directions of spread; the corresponding eigenvalues $\lambda_i$ describe the spread about these axes: the principal angular spreads can be approximated as $\theta_i = 2\text{arctan}\left(\sqrt{\lambda_i}\right)$. The spread amplitudes are listed in descending order and the axes form an orthonormal coordinate system. The rotation matrix from the Rodrigues space coordinate system to the principal axes coordinate system simply is,

$$v = \begin{pmatrix} v_{00} & v_{01} & v_{02} \\ v_{10} & v_{11} & v_{12} \\ v_{20} & v_{21} & v_{22} \end{pmatrix}$$

To go further, the orientation distribution about the principal axes can be obtained by expressing the orientations in the principal axes coordinate system,[2]

$$R_p = vR$$

and (for each component) considering the angular part,

$$\theta_i = 2\text{atan}(R_{p_i})$$

Figure 5.2 is an example of a bi-dimensional anisotropic spread of orientations, with its principal axes and the orientation distributions along these axes.

The functions described in this section are available from file `ol_set_dep.[c,h]`.

`void ol_set_aniso` (*struct OL_SET* `set`, *double*** `v`, *double** `theta`) [Function]
This routine computes the anisotropy properties of an orientation *set*. *v* are the principal axes and *theta* are the principal amplitudes along these axes.

`void ol_q_aniso_theta` (*double** `q`, *double*** `v`, *double** `theta`) [Function]
This routine computes the amplitudes along axes *v* for orientation *q*. *v* are the principal axes. The angles are recorded into *theta[i]* with $i \in \{0, 1, 2\}$.

[1] The spread can be expressed in any coordinate system provided that it is centred in the orientation space. The principal axes $v_i$ are expressed in that coordinate system. For example, this can be applied to disorientations expressed in the reference coordinate system or in a crystal coordinate system (Section 4.5 [Misorientations], page 41).

[2] This can be done about any axes, not only the principal axes.

`void ol_set_aniso_thetadistrib` (*struct OL_SET* `set`, *double*** `v`, *double* `interwidth`, *double*** `distrib`, *double** `distribfirstinter`, *int** `distribinterqty`) [Function]

This routine computes the orientation spread distribution along axes *v*. The *n* orientations are stored in *R* (*R[0], ..., R[n-1]*). *v* are the principal axes, *interwidth* is the angle interval width for the distributions. *distrib#* are the function outputs: *distrib[i]* with $i \in \{0, 1, 2\}$ contains the intensities of the distributions, *distribfirstinter[i]* are the first angle value of the distribution (beginning of the intervals), and *distribinterqty[i]* are the number of intervals.

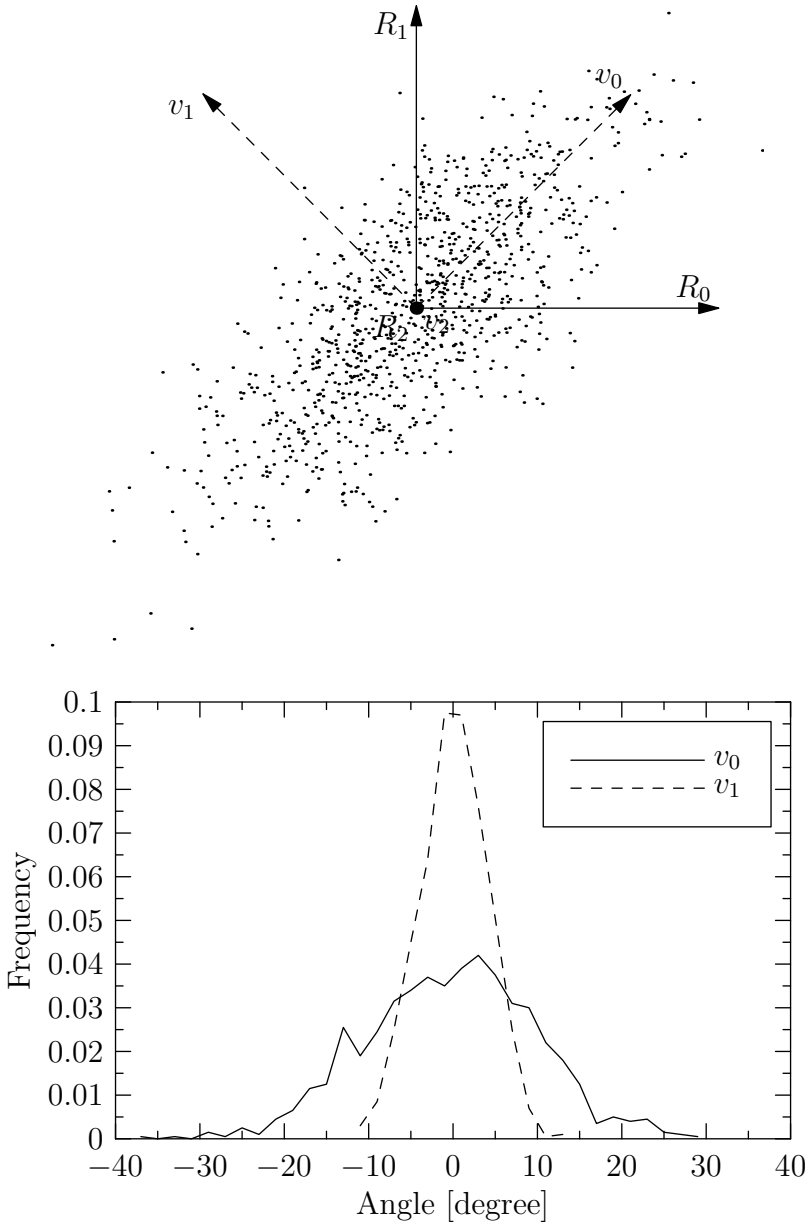



Figure 5.2: (Top) Bi-dimensional spread of orientations in the Rodrigues space and principal directions of spread $v_i$. The angles are $(\theta_0 = 9.98, \theta_1 = 3.96, \theta_2 = 0.00)$. (Bottom) Corresponding angle distributions along axes $v_0$ and $v_1$.

## 5.4 Orientation Spread Filtering

Sometimes, orientation spreads contain some highly-disoriented orientations that can be rejected.

A simple way is to define a maximum allowed disorientation from the average orientation, whose value can be chosen by multiplying the average disorientation by a given *factor*. After orientations are rejected, the spread average orientation can change, and so do the disorientations. Consequently, it can be necessary to apply this procedure iteratively.

For an anisotropic orientation spread, it is better to consider the disorientations along the three principal axes of the spread.

`int ol_set_filter` (*struct OL_SET* `set1`, *double* `factor`, *struct OL_SET* &*`set2`) [Function]

`int ol_set_filter_iter` (*struct OL_SET* `set1`, *double* `factor`, *struct OL_SET* &*`set2`) [Function]

These routines filter the orientation set *set1* using the parameter *factor*. The resulting orientation set is recorded in *set2*. The second routine applies this procedure iteratively. *set2* must be preallocated. The number of filtered orientations is returned.

The following functions are available from file `ol_set_dep.[c,h]`.

`int ol_set_filter_aniso` (*struct OL_SET* `set1`, *double* `factor`, *struct OL_SET* &*`set2`) [Function]

`int ol_set_filter_aniso_iter` (*struct OL_SET* `set1`, *double* `factor`, *struct OL_SET* &*`set2`) [Function]

These routines filter the orientation set *set1* using the parameter *factor*, consideing anisotropy . The resulting orientation set is recorded in *set2*. The second routine applies this procedure iteratively. *set2* must be preallocated. The number of filtered orientations is returned.

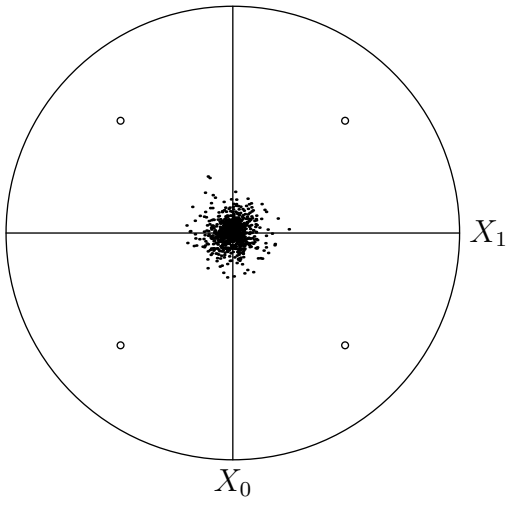



Figure 5.3: Case of an orientation spread with 4 highly-disoriented orientations (small circles) which are rejected by filtering with a *factor* of 5.

## 5.5 Examples

### 5.5.1 Mean Orientation

This example (available from directory `examples/ex6`) illustrates how to compute an average orientation. The 100 orientations (Euler angles) used are those output by the example of Section 3.3.2 [Generation Examples Spread], page 33.

```
#include<stdio.h>
#include<stdlib.h>
#include"ol/ol_nodep.h"
int
main (void)
{
  int i;
  struct OL_SET Set = ol_set_alloc (200);
  double *e = ol_e_alloc ();
  double *qm = ol_q_alloc ();
  FILE *in = fopen ("ex6.in", "r");
  FILE *out = fopen ("ex6.out", "w");

  for (i = 0; i < 200; i++)
    {
      ol_e_fscanf (in, e);
      ol_e_q (e, Set.q[i]);
    }

  ol_set_mean_iter (Set, Set.q[0], qm);
  ol_q_e (qm, e);
  ol_e_fprintf (out, e, "%.12f");

  fclose (in);
  fclose (out);

  ol_e_free (e);
  ol_q_free (qm);
  ol_set_free (Set);

  return EXIT_SUCCESS;
}
```

Here is what the example provides (see Figure 5.4),

```
$ ex6
-86.685886440492 0.526928607579 86.882024311290
```

(which is 0.56 degree away from $(0, 0, 0)$).

### 5.5.2 Volume Fraction of Texture Components

This example (available from directory `examples/ex7`) illustrates how to compute volume fraction of texture components. The program is not written in the manual for the sake of conciseness.

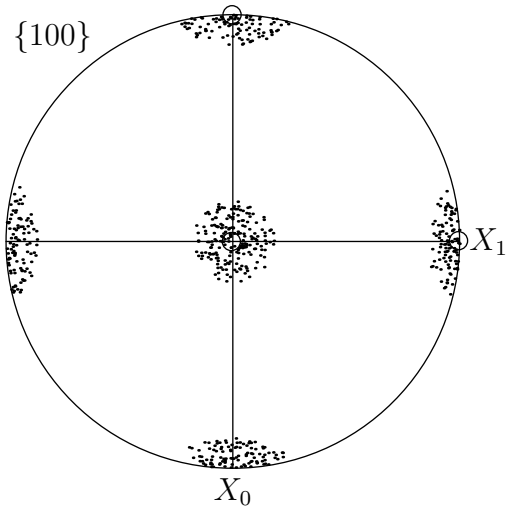



Figure 5.4: Set of 100 orientations (black points) and mean orientation (circle).

# 6 Orientation Mapping

This chapter describes functions specific to orientation maps. An *orientation map* is a set of orientations that are associated to a pre-defined pattern of coordinates. In the library, as illustrated of Figure 6.1, a rectangular pattern is assumed, with *xsize* data along axis $x$ and *ysize* data along axis $y$. The coordinates run from 0 to $xsize - 1$ and 0 to $ysize - 1$, respectively, by step of 1. The physical distance (e.g. in micrometer) from one point to the other is the same along both axes and is called *stepsize*. Every point is attributed a square zone of influence in which properties are assumed to be constant. Depending on the technique used to build the map, there may be various types of data available for every point. Here, two variables are used. The first one is *id*, which specifies whether the orientation is defined or not: $= 1$ if defined, $= 0$ else. The second is $q$, which is the orientation recorded as a quaternion. As a map is usually represented by an image, each orientation can be assigned a *rgb* colour made of three variables for the red, green and blue levels, ranging from 0 to 255.

In the library, an orientation map is defined by an `OL_MAP` structure. It contains six components, which are the variables defined above. The structure is very simple and looks like this,

```
typedef struct
{
  size_t         xsize;
  size_t         ysize;
  double         stepsize;
  unsigned int** id;
  double***      q;
  int***         rgb;
} OL_MAP;
```

For example, the point of coordinates `(i, j)` has the orientation `q[i][j]`, which is defined if `id[i][j] = 1`. It has colour *rgb[i]* (if defined).

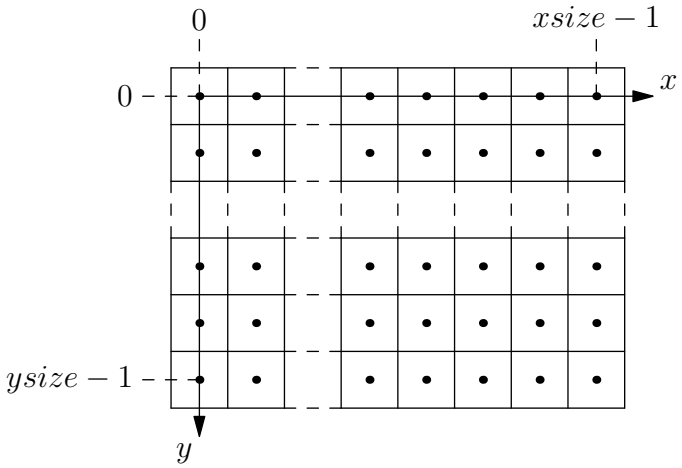



Figure 6.1: An orientation map. Note the positions of the axes. The orientations are represented by the dots and their zone of influence by the squares.

The functions described in this chapter are available from files `ol_map.[c,h]`.

## 6.1 Allocation

An orientation map is represented by an `OL_MAP` structure.

`struct OL_MAP ol_map_alloc` (*int* `xsize`, *int* `ysize`, *double* `stepsize`) [Function]
`void ol_map_free` (*struct OL_MAP* `map`) [Function]

The first routine creates an orientation map of size *xsize* × *ysize*; the step size is set to *stepsize*. The orientations are set as undefined: `id[i][j] = 0` and their colours to white: `rgb[i][j][k] = 255`.

The second frees a previously allocated orientation map *map*.

## 6.2 Copying

`void ol_map_memcpy` (*struct OL_MAP* `map1`, *struct OL_MAP** *&*`map2`) [Function]

This function copies an orientation map *map1* into an orientation map *map2*. *map2* must be preallocated and different from *map1*.

`void ol_map_submap` (*struct OL_MAP* `map1`, *int* `x0`, *int* `y0`, *int* `xsize`, *int* `ysize`, *struct OL_MAP** *&*`map2`) [Function]

This function copies part of an orientation map *map1*, made of *xsize*×*ysize* orientations from (*x0*, *y0*), into an orientation map *map2*. *map2* must be preallocated and different from *map1*.

## 6.3 Geometrical Transformation

Sometimes it is needed to process an orientation map like an image, e.g. to apply rotation, scaling, distortion, etc. Such *transformations* can be expressed through the *transformation gradient* $F$. Let $X = (x, y)$ be the old position of a point of the map. The position after transformation, called $X' = (x', y')$, is obtained as follows,

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} F_{xx} & F_{xy} \\ F_{yz} & F_{yy} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} S_x \\ S_x \end{pmatrix} \qquad \text{with} \qquad F = \frac{\partial X'}{\partial X}$$

As illustrated on Figure 6.2, the new map is so that the transformed map is exactly contained within it. The information of the transformed map are recorded into this regular map. The points that do not fall into the transformed map are set as undefined. $S$ is the shift vector so that the top-left point of the new map is translated at position $(0, 0)$.

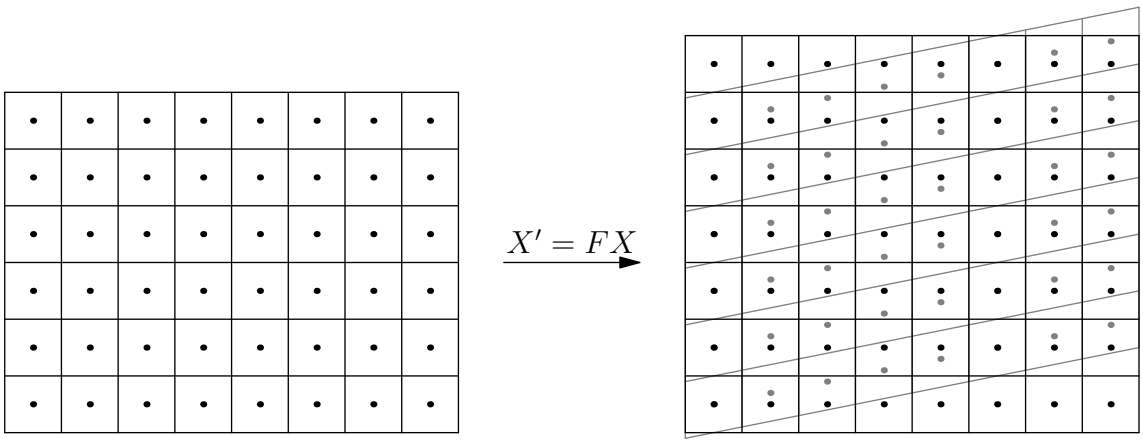



Figure 6.2: Transformation of an orientation map with $F_{xx} = 1$, $F_{xy} = 0$, $F_{yx} = 0.2$, $F_{yy} = 1$. On the left is the initial map. On the right is in gray the transformed map, and in black the new map.

`void ol_map_transformparam` (*struct OL_MAP* `map`, *double*** `F`, *int** `&xsize`, *int** `&ysize`, *double** `S`) [Function]

This routine returns the main parameters of the map resulting from the transformation $F$ of *map*. *xsize* and *ysize* are its sizes and $S$ is the shift vector.

`void ol_map_transform` (*struct OL_MAP* `map`, *double*** `F`, *int* `xsize`, *int* `ysize`, *double** `S`, *struct OL_MAP** `&map2`) [Function]

This routine transforms an orientation map *map* according to $F$. $S$ is the shift vector, *xsize* and *ysize* are the sizes of the new map. The new map is recorded into *map2*.

## 6.4 Reading and Writing

### 6.4.1 Native Format

Here are functions to read and write orientation maps at the Orientation Library native format.

`int ol_map_fscanf` (*FILE** `stream`, *struct OL_MAP** `&map`) [Function]

This routine reads formatted data from the stream *stream* into map *map*. *map* does not need to be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_map_fprintf` (*FILE* `stream`, struct OL_MAP `map`, char* `format`*) [Function]

This routine writes *map* to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

### 6.4.2 External Software Support

Because orientation maps are usually obtained by softwares which do not have the same file format than the library, some support functions are provided. Here are functions to read and write orientation maps at the commercial HKL Channel 5 software – in text format.

`int ol_map_fscanf_ch5` (*FILE* `stream`, struct OL_MAP* &`map`*) [Function]

This routine reads formatted data at the HKL Channel 5 format from stream *stream*[1] into map *map*. The file must be written at the default format. *map* does not need to be preallocated. The function returns a positive value for success and `EOF` if there was a problem reading from *stream*.

`int ol_map_fprintf_ch5` (*FILE* `stream`, struct OL_MAP `map`, char* `format`*) [Function]

This routine writes *map* at the HKL Channel 5 format to the stream *stream* using the format specifier *format*, which should be one of the `%g`, `%e` or `%f` formats. The function returns a positive value for success and a negative value if there was a problem writing to *stream*.

## 6.5 Printing as a PNG Image

Here are functions to write an image of a *map*. It is at the `PNG` (Portable Network Graphics) format, a bitmapped image format that employs lossless data compression.

`int ol_map_png` (*struct OL_MAP `map`, char* `filename`*) [Function]

`int ol_map_rgb_png` (*size_t `xsize`, size_t `ysize`, unsigned int*** `rgb`, char* `filename`*) [Function]

These routines write a *map* as a PNG image. The output file is named *filename*. The first routines takes the sizes and colours of the image from the *map* structure, while the second routine gives them directly.

[1] *stream* must be a seekable real file, but not a terminal or pipe.

# 7 References, Versions

## 7.1 References

Here are the documents which were used to build this library. You should keep in mind that some of them, although reference works, may contain mistakes – they are tagged *[errors found]*.

Books and papers related to orientations:


1. J. Hansen, J. Pospiech and K. Lücke, *Tables for Texture Analysis of Cubic Crystals*, Springer-Verlag, 1978.
2. V. Randle and O. Engler, *Introduction to texture analysis. Macrotexture, microtexture & orientation mapping*, Gordon and Breach Science Publishers, 2000. *[errors found]*
3. U.F. Kocks, C.N. Tomé and H.-R. Wenk, *Texture and Anisotropy*, Cambrige University Press, 2000. *[errors found]*
4. F.J. Humphreys, P.S. Bate and P.J. Hurley, *Orientation averaging of electron backscattered diffraction data*, Journal of Microscopy, Vol. 201, pp 50–58, 2001.
5. J.Ch. Glez and J.H. Driver, *Orientation distribution analysis in deformed grains*, Journal of Applied Crystallography, Vol. 34, pp 280–288, 2001. *[errors found]*
6. A. Morawiec, *Orientations and Rotations, Computations in crystallographic textures*, Springer, 2003.


See also:

- R. Quey, *HeRMeS Reference Manual - A program for microstructure characterization. Version 1.4*, École Nationale Supérieure des Mines de Saint-Etienne, France, 2008.

## 7.2 Versions

```
New in 2.0.3 (05 Nov 2014):
General:  small enhancements.
Description:  fixed up documentation for quaternion to Euler angles
conversion, fixed up code and documentation for Euler angles to
quaternion conversion for the special cases where phi=0 or 180 degrees.

New in 2.0.2 (29 jun 2008):
General:  manual corrections, small enhancements.

New in 2.0.1 (27 jun 2008):
General:  manual corrections, small enhancements.
Calculation:  added "shortcut" functions ol_e_e_disori and ol_e_e_disori.

New in 2.0.0 (19 jun 2008):
General:   new functions on orientation sets and maps ; modified  file
organization due to the library growth:  ol.c and ol.h replaced by files  for
the individual "modules" (or  chapter)  of  the  software:   ol_des,  ol_cal,
ol_gen, ol_set, ol_map.
Interactive  program:   new  way-of-use  (convenient  for  scripting);  new
functions.
Description:  added "shortcut" functions for  conversions;  code  and  manual
corrections for reading and writing function  (_fscanf,  _fprintf)  returned
values.
```

Generation: added functions to generate random (mis-)orientations.
Calculation: orientation averaging function moved to "Set", new functions for specimen symmetry; fixed misorientation functions with quaternions; renamed function ol_r_r_angle to ol_vect_vect_theta and new function ol_vect_vect_rtheta.
Set: changed ol_q_mean to ol_set_mean; new orientation averaging function for large spreads; new functions for orientation spread filtering; new functions for anisotropic orientation spread study.
Map: new functions for orientation mapping together with support for external softwares (HKL Channel 5) and image printing (PNG format).

New in 1.2.0 (12 mar 2008):
Very first version of the Orientation Library interactive program; new functions for generation of misorientations (ol_nb_max_theta_mis, ol_nb_max_rtheta_mis); corrected / improved generation of orientations (ol_nb_r, ol_nb_max_rtheta); new functions for descriptor initialization (ol_*_set_this), new function for disorientation calculation (ol_g_g_disori); new allocation / printing functions for pole figures, improved memory allocation functions (tests included); renamed functions ol_nb_theta_max and ol_nb_rtheta_max to ol_nb_max_theta and ol_nb_max_rtheta; manual syntax corrections and enhancements.

New in 1.1.3 (05 feb 2008):
Many manual syntax corrections and enhancements; new conversion functions ol_R_q and ol_q_R; corrected generation of axis/angle of rotation (ol_nb_rtheta ...); changed order of arguments of function ol_g_m_quality; renamed functions ol_g_frame and ol_q_frame to ol_g_csys and ol_q_csys.

New in 1.1.2 (21 jan 2008):
Improved conversion of rotation matrix into Euler angles by accounting for Euler space degeneracy at phi=180 (ol_g_e, ol_e_e); improved conversion of quaternions into Euler angles by accounting for Euler space degeneracy at phi=0 and phi=180 (ol_q_e); new functions ol_g_stprojxy, ol_g_eaprojxy, ol_g_g_gmisori_ref, ol_q_q_qmisori_ref, ol_g_g_gmisori, ol_q_q_qmisori, ol_g_g_gmisori_cur, ol_q_q_qmisori_cur and ol_q_q_misori; various manual corrections and enhancements.

New in 1.1 (07 jan 2008):
The Orientation Library is distributed under GPL: (http://sourceforge.net/projects/orilib)! Plenty of changes and improvements everywhere. If you are moving from version 1.0, you should consider checking every routine you are using.

New in 1.0.1 (11 jul 2007):
Original full version.

# Appendix A Elements of Quaternion Algebra

- Definition of a quaternion

  A *quaternion* $q = (\rho, \lambda, \mu, \nu)$ is defined as follows:

  $$q = \rho + i\lambda + j\mu + k\nu \quad \text{with} \quad i^2 = j^2 = k^2 = ijk = -1$$

  It is composed of a scalar part $\rho$ and a vector part $(\lambda, \mu, \nu)$.

  Moreover, to describe space rotations, one are using *unit quaternions*:

  $$|q| = \sqrt{\rho^2 + \lambda^2 + \mu^2 + \nu^2} = 1$$

- Conjugate of a quaternion

  The *conjugate of a quaternion* is given by,

  $$q^c = (\rho, -\lambda, -\mu, -\nu)$$

- Inverse of a unit quaternion

  The *inverse of a unit quaternion* is equal to its conjugate,

  $$q^{-1} = q^c$$

- Sum of quaternions

  Let $q_1 = (\rho_1, \lambda_1, \mu_1, \nu_1)$ and $q_2 = (\rho_2, \lambda_2, \mu_2, \nu_2)$ be two quaternions. The *sum of* $q_1$ *and* $q_2$, called $q_3 = (\rho_3, \lambda_3, \mu_3, \nu_3)$, is,

  $$q_3 = q_1 + q_2 = \left| \begin{array}{l} \rho_3 = \rho_1 + \rho_2 \\ \lambda_3 = \lambda_1 + \lambda_2 \\ \mu_3 = \mu_1 + \mu_2 \\ \nu_3 = \nu_1 + \nu_2 \end{array} \right.$$

  The sum of quaternions is associative and commutative.

- Product of quaternions

  Let $q_1 = (\rho_1, \lambda_1, \mu_1, \nu_1)$ and $q_2 = (\rho_2, \lambda_2, \mu_2, \nu_2)$ be two quaternions. The *product of* $q_1$ *and* $q_2$, called $q_3 = (\rho_3, \lambda_3, \mu_3, \nu_3)$, is,

  $$q_3 = q_1 q_2 = \left| \begin{array}{l} \rho_3 = \rho_1\rho_2 - \lambda_1\lambda_2 - \mu_1\mu_2 - \nu_1\nu_2 \\ \lambda_3 = \rho_1\lambda_2 + \lambda_1\rho_2 + \mu_1\nu_2 - \nu_1\mu_2 \\ \mu_3 = \rho_1\mu_2 - \lambda_1\nu_2 + \mu_1\rho_2 + \nu_1\lambda_2 \\ \nu_3 = \rho_1\nu_2 + \lambda_1\mu_2 - \mu_1\lambda_2 + \nu_1\rho_2 \end{array} \right.$$

  The product of quaternions is associative and distributive, but not commutative (in the general case):

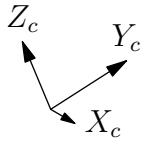
  $$\left\{ \begin{array}{l} (q_1 q_2) q_3 = q_1 (q_2 q_3) \\ q_0(q_1 + q_2) = q_0 q_2 + q_0 q_1 \text{ and } (q_1 + q_2) q_0 = q_1 q_0 + q_2 q_0 \\ q_1 q_2 \neq q_2 q_1 \end{array} \right.$$

# Appendix B Cubic Symmetry Operators

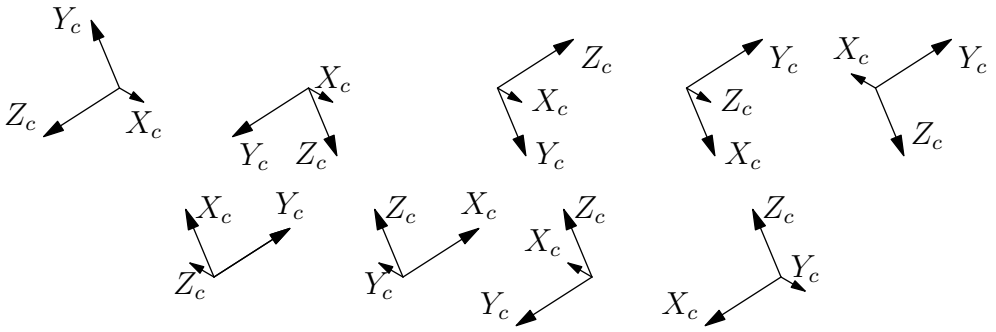
## B.1 Coordinate Systems

Here are illustrated the coordinate system positions of the 24 crystallographically-related orientations,

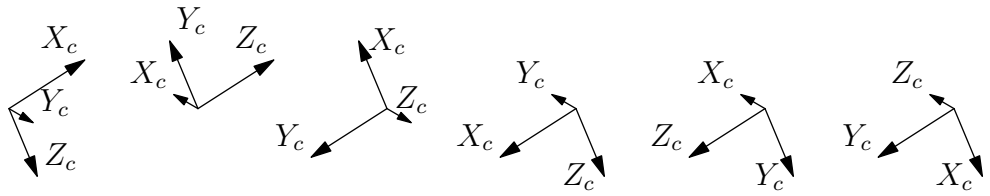
- The identity,



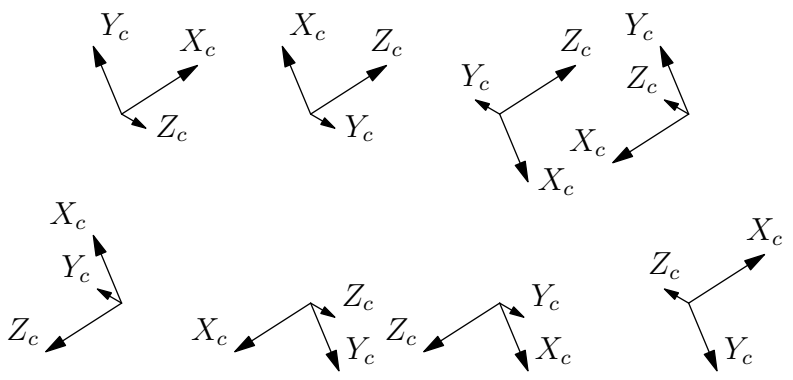

- the three rotations of 90 degrees about each of the three $\langle 100 \rangle$ ([100], [010] and [001], successively),




- one rotation of 180 degrees about each of the six $\langle 110 \rangle$ ([110], [011], [101], $[\bar{1}10]$, $[0\bar{1}1]$ and $[10\bar{1}]$, successively),




- the two rotations of 120 degrees about each of the four $\langle 111 \rangle$ ([111], $[\bar{1}11]$, $[1\bar{1}1]$ and $[11\bar{1}]$, successively),



## B.2 Rotation Matrices

Let an orientation be described by a rotation matrix $g$. The crystallographically-related orientations can be calculated as follows,

$$g_i = T_i g$$

where $T_i$ is ('$-1$' is written as $\overline{1}$ for the sake of compactness),

- The identity,

$$T_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- the three rotations of 90 degrees about each of the three $\langle 100 \rangle$ ([100], [010] and [001], successively),

$$T_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & \overline{1} & 0 \end{pmatrix} \quad T_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \overline{1} & 0 \\ 0 & 0 & \overline{1} \end{pmatrix} \quad T_4 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & \overline{1} \\ 0 & 1 & 0 \end{pmatrix}$$

$$T_5 = \begin{pmatrix} 0 & 0 & \overline{1} \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad T_6 = \begin{pmatrix} \overline{1} & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \overline{1} \end{pmatrix} \quad T_7 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ \overline{1} & 0 & 0 \end{pmatrix}$$

$$T_8 = \begin{pmatrix} 0 & 1 & 0 \\ \overline{1} & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad T_9 = \begin{pmatrix} \overline{1} & 0 & 0 \\ 0 & \overline{1} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad T_{10} = \begin{pmatrix} 0 & \overline{1} & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- one rotation of 180 degrees about each of the six $\langle 110 \rangle$ ([110], [011], [101], $[\overline{1}10]$, $[0\overline{1}1]$ and $[10\overline{1}]$, successively),

$$T_{11} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & \overline{1} \end{pmatrix} \quad T_{12} = \begin{pmatrix} \overline{1} & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad T_{13} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & \overline{1} & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$T_{14} = \begin{pmatrix} 0 & \overline{1} & 0 \\ \overline{1} & 0 & 0 \\ 0 & 0 & \overline{1} \end{pmatrix} \quad T_{15} = \begin{pmatrix} \overline{1} & 0 & 0 \\ 0 & 0 & \overline{1} \\ 0 & \overline{1} & 0 \end{pmatrix} \quad T_{16} = \begin{pmatrix} 0 & 0 & \overline{1} \\ 0 & \overline{1} & 0 \\ \overline{1} & 0 & 0 \end{pmatrix}$$

- the two rotations of 120 degrees about each of the four $\langle 111 \rangle$ ([111], $[\overline{1}11]$, $[1\overline{1}1]$ and $[11\overline{1}]$, successively),

$$T_{17} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad T_{18} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad T_{19} = \begin{pmatrix} 0 & 0 & \overline{1} \\ \overline{1} & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad T_{20} = \begin{pmatrix} 0 & \overline{1} & 0 \\ 0 & 0 & 1 \\ \overline{1} & 0 & 0 \end{pmatrix}$$

$$T_{21} = \begin{pmatrix} 0 & 0 & 1 \\ \overline{1} & 0 & 0 \\ 0 & \overline{1} & 0 \end{pmatrix} \quad T_{22} = \begin{pmatrix} 0 & \overline{1} & 0 \\ 0 & 0 & \overline{1} \\ 1 & 0 & 0 \end{pmatrix} \quad T_{23} = \begin{pmatrix} 0 & 0 & \overline{1} \\ 1 & 0 & 0 \\ 0 & \overline{1} & 0 \end{pmatrix} \quad T_{24} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & \overline{1} \\ \overline{1} & 0 & 0 \end{pmatrix}$$

## B.3 Quaternions

Let an orientation be described by a quaternion $q$. The crystallographically-related orientations can be calculated as follows,

$$q_i = qU_i$$

where $U_i$ is,

- The identity,

$$U_1 = \quad (1, \quad 0, \quad 0, \quad 0)$$

- the three rotations of 90 degrees about each of the three $\langle 100 \rangle$ ([100], [010] and [001], successively),

$$\begin{array}{rrrrr}
U_2 = & ( & 1/\sqrt{2}, & 1/\sqrt{2}, & 0, & 0) \\
U_3 = & ( & 0, & 1, & 0, & 0) \\
U_4 = & ( & -1/\sqrt{2}, & 1/\sqrt{2}, & 0, & 0) \\
U_5 = & ( & 1/\sqrt{2}, & 0, & 1/\sqrt{2}, & 0) \\
U_6 = & ( & 0, & 0, & 1, & 0) \\
U_7 = & ( & -1/\sqrt{2}, & 0, & 1/\sqrt{2}, & 0) \\
U_8 = & ( & 1/\sqrt{2}, & 0, & 0, & 1/\sqrt{2}) \\
U_9 = & ( & 0, & 0, & 0, & 1) \\
U_{10} = & ( & -1/\sqrt{2}, & 0, & 0, & 1/\sqrt{2})
\end{array}$$

- one rotation of 180 degrees about each of the six $\langle 110 \rangle$ ([110], [011], [101], $[\bar{1}10]$, $[0\bar{1}1]$ and $[10\bar{1}]$, successively),

$$\begin{array}{rrrrr}
U_{11} = & (0, & 1/\sqrt{2}, & 1/\sqrt{2}, & 0) \\
U_{12} = & (0, & 0, & 1/\sqrt{2}, & 1/\sqrt{2}) \\
U_{13} = & (0, & 1/\sqrt{2}, & 0, & 1/\sqrt{2}) \\
U_{14} = & (0, & -1/\sqrt{2}, & 1/\sqrt{2}, & 0) \\
U_{15} = & (0, & 0, & -1/\sqrt{2}, & 1/\sqrt{2}) \\
U_{16} = & (0, & 1/\sqrt{2}, & 0, & -1/\sqrt{2})
\end{array}$$

- the two rotations of 120 degrees about each of the four $\langle 111 \rangle$ ([111], $[\bar{1}11]$, $[1\bar{1}1]$ and $[11\bar{1}]$, successively),

$$\begin{array}{rrrrrrrrr}
U_{17} = & (1/2, & 1/2, & 1/2, & 1/2) & U_{18} = & (-1/2, & 1/2, & 1/2, & 1/2) \\
U_{19} = & (1/2, & -1/2, & 1/2, & 1/2) & U_{20} = & (-1/2, & -1/2, & 1/2, & 1/2) \\
U_{21} = & (1/2, & 1/2, & -1/2, & 1/2) & U_{22} = & (-1/2, & 1/2, & -1/2, & 1/2) \\
U_{23} = & (1/2, & 1/2, & 1/2, & -1/2) & U_{24} = & (-1/2, & 1/2, & 1/2, & -1/2)
\end{array}$$

## B.4 Miller Indices

Let an orientation be described by Miller indices $(hkl)\,[uvw]$. The crystallographically-related orientations can be calculated through permutations of the indices, $p_i$, as follows,

- The identity,

$$p_1 : (hkl)\,[uvw]$$

- the three rotations of 90 degrees about each of the three $\langle 100 \rangle$ ([100], [010] and [001], successively),

$$\begin{array}{lll} p_2 : (hk\bar{l})\,[uv\overline{w}] & p_3 : (h\overline{k}l)\,[u\overline{v}w] & p_4 : (h\bar{l}k)\,[u\overline{w}v] \\ p_5 : (\bar{l}kh)\,[\overline{w}vu] & p_6 : (\overline{h}k\bar{l})\,[\overline{u}v\overline{w}] & p_7 : (lk\overline{h})\,[wv\overline{u}] \\ p_8 : (k\overline{h}l)\,[v\overline{u}w] & p_9 : (\overline{h}\overline{k}l)\,[\overline{u}\overline{v}w] & p_{10} : (\overline{k}hl)\,[\overline{v}uw] \end{array}$$

- one rotation of 180 degrees about each of the six $\langle 110 \rangle$ ([110], [011], [101], $[\overline{1}10]$, $[0\overline{1}1]$ and $[10\overline{1}]$, successively),

$$\begin{array}{lll} p_{11} : (kh\bar{l})\,[vu\overline{w}] & p_{12} : (\overline{h}lk)\,[\overline{u}wv] & p_{13} : (l\overline{k}h)\,[w\overline{v}u] \\ p_{14} : (\overline{k}hl)\,[\overline{v}uw] & p_{15} : (\overline{h}l\overline{k})\,[\overline{u}w\overline{v}] & p_{16} : (\overline{l}kh)\,[\overline{w}vu] \end{array}$$

- the two rotations of 120 degrees about each of the four $\langle 111 \rangle$ ([111], $[\overline{1}11]$, $[1\overline{1}1]$ and $[11\overline{1}]$, successively),

$$\begin{array}{llll} p_{17} : (klh)\,[vwu] & p_{18} : (lhk)\,[wuv] & p_{19} : (\overline{l}\overline{h}k)\,[\overline{w}\overline{u}v] & p_{20} : (\overline{k}l\overline{h})\,[\overline{v}w\overline{u}] \\ p_{21} : (l\overline{h}\overline{k})\,[w\overline{u}\overline{v}] & p_{22} : (\overline{k}lh)\,[\overline{v}\overline{w}u] & p_{23} : (\bar{l}h\overline{k})\,[\overline{w}u\overline{v}] & p_{24} : (k\overline{l}\overline{h})\,[v\overline{w}\overline{u}] \end{array}$$

# Appendix C GNU General Public License

GNU General Public License

Version 3, 29 June 2007



## Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

   The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

   The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms

that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

   When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

   a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

    If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

    The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

    If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

    Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

    THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRIT-

ING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

    IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

    If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

## END OF TERMS AND CONDITIONS

## How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

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 3 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, see http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘`show w`’ and ‘`show c`’ should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `http://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `http://www.gnu.org/philosophy/why-not-lgpl.html`.

# Appendix D Function Index