SimulTrough Library User's Guide

December 2007

Contents

Introduction

SimulTrough is a C++ library that performs optical simulations of solar parabolic trough collectors, with a receiver pipe in the focal line. This technology is seen as one of the most promising in the field of renewable energy. The SimulTrough library is specialized in the simulation of parabolic troughs, with the possibility of setting the optical properties of the mirror and of the receiver, choosing the direction of incidence of the sun (simulating non-orthogonal incidence or tracking errors) and simulating defects of the system such as an out-of-focus receiver or deformations of the mirror. This can be a useful instrument to predict the real optical efficiency of a system in working conditions and to study the tolerance to defects and the importance of optical quality.

The library is distributed under the GNU General Public License (see Appendix A). It can be freely distributed and modified, but not included in proprietary programs. Of course, the library is distributed without any warranty, without even the implied warranty of merchantability or fitness for a particular purpose.

1. General description

The library uses a recursive ray-tracing technique to perform optical simulations of a system composed of parabolic trough collectors, with the receiver pipe of circular section in the focus. The receiver pipe can have different protective layers, such as the glass covering in the classical receiver; the only request is that the layers are circular and concentrical around the focus. The number of protective layers is arbitrary, even though the cases of practical interest are usually limited to the evacuated tube with a glass covering, and a tube without any covering.

All the geometrical and optical properties of the trough can be set by the user. Each surface of the system must be endowed with a reflective law, depending on the angle of incidence and on the wavelength. For each medium, a real index of refraction, depending on the wavelength, and an absorption law, depending on the length of the path through the medium and on the wavelength, must be defined. Even if, for an homogeneous medium, these functions are related (the reflectivity and the absorption can be obtained from the indices of refraction), in the library the three properties are kept independent, in order to simulate more general situations, such as the presence of thin films: e. g., an anti-reflex layer on the glass has a reflection law that cannot be obtained from the index of refraction of the glass. Moreover, if the wavelength dependence is ignored – as it often happens – the optical laws can be defined calculating an average law on the solar spectrum, and this law cannot be derived from the index of refraction of the medium. The independence of the absorption law from the index of refraction make the complex refractive index useless; so, the given index must be real, ignoring the imaginary part. It must be stressed that a non-negligible imaginary part of the refraction index completely invalidates the ray-tracing analysis, so, in the tractable cases, the choice of a real index is not a restriction.

The simulation is performed by shooting a certain number of solar beams on the collector, at regular intervals on the collector width; the number of solar beams can be set by the user. A solar beam is a group of rays with the angular distribution of the sun rays, with the energy modulated in a limb-darkened model; 90% of rays are used to simulate the solar disk, and 10% are used for the halo. The solar wavelength spectrum can also be considered, if the optical properties depend significantly on the wavelength; in this case, for each “ray” of the solar beam a certain number of rays of different wavelength are shot, with the energy distribution of the solar spectrum.

The beams can be shot from arbitrary directions with respect to the collector, so as to simulate the inclination of the rays incident on the collector**,** or the tracking error.

Several kinds of defects of the system can be simulated:

- Defocalisation: the receiver centre is not exactly in the focus, e. g. because of thermal or gravitational deformations.
- Longitudinal random error on the inclination of the mirror: Gaussian errors on the inclination of the mirror, in the direction parallel to the receiver.
- Transverse random error on the inclination of the mirror: Gaussian errors on the inclination of the mirror, in the transversal section (errors on the parabolic profile). This can be different from the longitudinal error.
- Large-scale transverse errors: large-scale deformations of the parabolic profile, because of thermal stress, or the effect of supports, or errors in the constructions. The profile error must be given as a Fourier series.

The simulation gives the distribution of the solar radiation on the receiver surface, for a unitary incident direct radiation, and the total collection efficiency. The radiation absorbed in the glass layer, if such a layer is present, can also be obtained.

2. Files

Four files are given: `SimulTrough.h` (header file), `SimulTrough.cpp` (library source file), `opt_fun_definition.h` (auxiliary file where the optical parameters are already defined, for simplicity of use), and `SimulTrough_UserGuide.pdf` (this document).

In all the programs where the library is used, the header `SimulTrough.h` must be included by adding the following command at the beginning of the program

```
#include”(path)\SimulTrough.h”
```

or, if the file is copied in the standard header directory,

```
#include<SimulTrough>
```

If one chooses to use the optical function defined in `opt_fun_definition.h`, possibly modified by the user, this file must also be included with the command

```
#include”(path)\opt_fun_definition.h”
```

or

```
#include<opt_fun_definition>
```

Moreover, the file `SimulTrough.cpp` must be included in the project.

The file `SimulTrough.cpp` requires to include the file `SimulTrough.h`; the first code line of the file is

```
#include”SimulTrough.h”
```

which of course works only if the header file and the library source file are placed in the same directory. Otherwise, this line must be changed, adding the path of the header file.

The file `SimulTrough.cpp` can be compiled once to create a library and then linked after the compilation when needed, without including the source file in the project and without re-compiling it every time. The choice of supplying the source code is due to compatibility reasons (the source file should work with all compilers and all operative systems), and it is meant to give the user the possibility of modifying the file.

3. A first simple example

This program is a first example of a simulation, dealing with the most simple case. More accurate and systematic explanation will be given later.

```
#include<iostream>
#include<cmath>

#include"SimulTrough.h"
#include"opt_fun_definition.h"

using namespace std;

int main() {

   OPT_FUN_INIT;

   int NumSurface = 3;

   double RadiiReceiver[] = {0.065, 0.062, 0.035};

   double Focus = 1.8,
         MirrorWidth = 3.0,
         FreeSpace = 0.05;

   opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
         RadiiReceiver, ReflectMir, ReflectSurfaces, RefInd,
         Absorption);

   simul.simulate( 0, 1, 0 );
   cout<<endl<<endl;

   for (int i=0; i<72; i++) cout<<"Energy collected between "<<
         5*i<<"° and "<<5*(i+1)<<"° = "<<
         simul.get_absorber(i)<<endl;

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;

}
```

In the first lines, the library header (`SimulTrough.h`) and the file with the definitions of the optical functions (`opt_fun_definition.h`) are included.

The command `OPT_FUN_INIT`, at the beginning of the program, calls a macro that initializes the optical properties; the macro is defined in `opt_fun_definition.h`. This macro defines the variables `ReflectMir, ReflectSurfaces, RefInd, Absorption,` that will be used later; they are pointers to the optical functions. More details about the optical functions will be given in the next section.

The following command lines (the three variable declarations) defines the geometrical properties of the system:

- The number `NumSurface` of concentric surfaces around the receiver, including the absorbing surface. A surface is a separation between two media: for example, the classical receiver with a glass covering has 3 surfaces (receiving surface, internal face of the glass, external face of the glass); this is the case considered in the example. A naked receiver would have 1 surface.
- The radii of the concentric surfaces, given as a vector called `RadiiReceiver,` in order from the most external to the most internal (the absorbing surface). This vector (or pointer) must of course have a number of elements equal to `NumSurface`. In this example, the diameter of the absorber is 7 cm (radius 0.035

m), and the glass covering has an external diameter of 13 cm and a thickness of 3 mm (the two radii are 0.065 m and 0.062 m).
- The focal distance (`Focus`) of the system (in this case, 1.8 m);
- The semi-width of the parabolic collector (`MirrorWidth`) (in this case, the collector has a total width of 6 m);
- The semi-width of the possible free space at the vertex of the parabola (`FreeSpace`), between the two semi-parabolas, often present in trough systems (in this case, the two halves of the mirror are separated by a distance of 10 cm, so the variable `FreeSpace` has the value 0.05).

All these geometrical measures must of course be given in the same unity (meters in this case).

In the next line (`opt_simulation ...`) an object of the class `opt_simulation` is built, called `simul`. This object is a complete description of the system, and it must be built with the constructor passing the parameters in the order shown: first the five geometrical data (`Focus, MirrorWidth, FreeSpace, NumSurface, RadiiReceiver`) and then the optical functions (`ReflectMir, ReflectSurfaces, RefInd, Absorption`), in this case defined by the macro `OPT_FUN_INIT`. These nine parameters are always required in all cases. There are other optional parameters that can be given, to introduce defects or to set numerical parameters: they will be described later.

The following line (`simul.simulate(…)`) performs the simulation, for the system defined in the object `simul`. The position of the sun must be passed to the function; it is identified by a vector (*x*, *y*, *z*) that points towards the centre of the sun with respect to the collector. The *z* axis is on the vertex line, the *y* axis is orthogonal to the collecting plane of the collector, and the *x* axis is orthogonal to both. The parabolic section lays in the *xy* plane, and the receiver is parallel to the *z* axis. In the example shown above, the sun is exactly orthogonal above the collector: only the *y* component of the vector is different from 0. The vector is passed to the function `simulate` as three `double` parameters, in the order *x*, *y*, *z* (in this case 0, 1, 0).

During the simulation, the distribution of the radiation absorbed by the receiver is computed. The receiver circumference is divided into angular sectors (their default number is 72, and each sector has a width of 5°) and the energy absorbed in each sector is saved. The sectors are enumerated from 0 to 71 in counterclockwise direction, starting from the point opposite to the collector, on the dark side of the receiver. At the end of the simulation, the function `simul.get_absorber(i)` returns the energy absorbed in the sector numbered with `i`. In the `for` cycle `i` goes from 0 to 71, so the distribution of the absorbed energy is printed on the screen. At the end, the total optical efficiency is printed; it is read with the function `simul.get_efficiency()`.

Launching the program, the output window will show

```
Setting solar position ....... OK

x0 = -3  OK
x0 = -2.97     OK
x0 = -2.94     OK

(...)

x0 = 3   OK
```

```
Energy collected between 0° and 5° = 0.000559756
Energy collected between 5° and 10° = 0.000537184
Energy collected between 10° and 15° = 0.000445569

(...)

Energy collected between 355° and 360° = 0.000547798

Efficiency = 0.774441
```

The lines `x0 =` ... are printed during the simulation to show its progress (this feature can be disabled). `x0` varies from `-MirrorWidth` to `MirrorWidth`. The following lines (`Energy collected between` ...) are printed in the `for` cycle, and show the energy absorbed in each sector; the sum of these values is the total optical efficiency.

4. Optical functions

As shown in the preceding section, the main structure of the library is the class `opt_simulation`. This class is created before performing any simulation, and it contains all the data on the collector. When creating the class, all the geometric data must be given, and also a list of pointers to the optical functions of the various surfaces and media. So, before the simulation one has to define the optical functions. This section shows how to perform this task. It can seem quite laborious, but probably it will almost never be done in the actual work; this section is aimed to show the way in which the library deals with optical parameters, and to show how to use properly the `opt_fun_definition.h` file or a similar created by the user. The reader who wants to see more examples and practice before changing the optics may skip this section and go to the following, initializing the optical functions as shown in the example of Section 3 (the header `opt_fun_definition.h` must be included), and returning here later.

The procedure to inizialize the optical properties requires three steps:

a) Writing the optical functions;
b) Associating the optical functions with function pointers;
c) Passing the pointers to the `opt_simulation` constructor.

a) There are three kind of optical functions: reflectivity functions, associated with each surface, refractive indices and absorption functions, associated with each medium.

A function of reflectivity must have two `double` parameters: the cosine of the angle of incidence and the wavelength of the light. All the wavelength dependences must be expressed with the wavelength measured in Ångstrom (an Ångstrom is 10^{-10} m; the wavelength value in the solar spectrum goes from 3000 to 25000 Å). The function must return a `double`, the fraction of energy of the ray reflected. In total `2*NumSurface` reflectivity functions must be defined (`NumSurface`, as in the example of the previous section, is the number of concentric surfaces around the focus): one function for the mirror and the others for the concentric surfaces. 2 reflectivity functions are associated with each concentric surface (except the absorber), one for each side. As an example, in the classical case of a glass covered receiver, the functions are 6: one for the mirror, one for the receiver surface, two for the external face of the glass (one for rays coming from the air, and one for rays coming from the glass), and two for the internal face of the glass[1].

When a ray hits the mirror, the fraction which is not reflected is considered lost. When the ray hits the absorber, the fraction which is not reflected is considered absorbed and converted to heat. When the ray hits one of the concentric surfaces around the absorber (e. g. a face of the glass) the fraction not reflected is considered refracted.

A function of refractive index has one `double` parameter, the wavelength (in Ångstrom), and it returns a `double`, the refractive index. The media in which the light can travel are `NumSurface`: the air, plus the `NumSurface-1` media contained between the concentric surfaces. In the case of the classical receiver, the media are air, glass, and the vacuum. So, a number `NumSurface` of refractive index functions must be defined. The refraction index must be real, since the ray tracing method fails for a high imaginary part of the refraction index; the absorption is considered in other optical functions.

A function of absorption receives two `double` parameters, the length of the travel of the ray in the medium and the wavelength (in Ångstrom), and it returns a `double`, the fraction of energy of the emerging ray (*not* the absorbed fraction). The number of absorption functions is the same as the number of media (`NumSurface`, as in the case of refraction indices).

b) Once all the functions have been defined, they must be assigned to function pointers, in order to be passed to the simulation. A pointer to `double(double, double)` will contain the mirror reflectivity; an array of pointers of the same type with `2*NumSurface-1` elements will contain the functions of reflectivity of the concentric surfaces, in order from the outermost to the inmost, with the reflectivity for rays from the outside coming before the reflectivity for rays from the inside. For example, for a classical evacuated receiver the order is:

Element 0: external face of the glass, ray from the air;
Element 1: external face of the glass, ray from the glass;
Element 2: internal face of the glass, ray from the glass;
Element 3: internal face of the glass, ray from the vacuum;
Element 4: absorbing surface.

Another array of pointers to `double(double)` of `NumSurface` elements will contain the refractive indices, from the outermost to the inmost (in the classical case: air, glass, vacuum), and a third array of pointers to `double(double, double)` will contain the absorption functions, in the same order.

c) These pointers are then passed to the constructor. In the ray-tracing process, during the simulation, these optical functions will be used each time a ray crosses a medium or hits a surface.

In the example of Section 3, the definition of the optical functions was in the file `opt_fun_definition.h`, while the definition of the pointers was hidden in the `OPT_FUN_INIT` macro. So, once the definition has been done a first time, the process of initialization can be done in a single line.

[1] Of course the two functions associated with the same surface are not independent, but the implementation of the program requires that they are given separately.

Now we will show step by step the definition of the optical functions in the case of a classical receiver; at the same time, we will build the `opt_fun_definition.h` file.

In this case, we have `NumSurface=3`: the absorbing surface, the internal face of the glass, and the external face of the glass. So one must define 6 functions of reflectivity:

1) a function for the mirror: as an example, we consider a simple (and unrealistic) model of mirror, with reflectivity 1-0.1cos(θ), where θ is the incidence angle; the normal reflectivity is 0.9. We can define the simple function (independent of the wavelength `lambda`)

```
double MirrorReflectivity (double cosinc, double lambda) {

   return 1-0.1*cosinc;
}
```

2) a function for the reflectivity of the external face of the glass, when the ray comes from outer space (the air). As an example, we consider a glass without thin layer coverings, and with a law of reflection determined by classical wave optics, assuming an index of refraction 1.5, and we neglect the wavelength dependence:

```
double ReflectGlassExternOut (double cosinc, double lambda) {

   double R_perp = (cosinc-sqrt(1.25+cosinc*cosinc))/(cosinc+
                  sqrt(1.25+cosinc*cosinc));

   double R_par = (2.25*cosinc-
         sqrt(1.25+cosinc*cosinc))/(2.25*cosinc+
         sqrt(1.25+cosinc*cosinc));

   return 0.5*(R_perp*R_perp+R_par*R_par);

}
```

`R_par` squared is the reflectivity for polarization parallel to the surface, while `R_perp` squared is for polarization in the orthogonal direction; the reflectivity for unpolarized light is the mean of the two.

3) a function for the reflectivity of the external face of the glass, when the ray comes from inner space (the glass). This function is the inverse of the above one, but it must also consider the possible total reflection:

```
double ReflectGlassExternIn (double cosinc, double lambda) {

   if ( cosinc*cosinc <= 0.5556)  return 1; // total reflection

   double R_perp = (cosinc-sqrt(-0.5556+cosinc*cosinc))/(cosinc
         +sqrt(-0.5556+cosinc*cosinc));

   double R_par = (cosinc-1.5*sqrt(-
          1.25+2.25*cosinc*cosinc))/(cosinc+
          1.5*sqrt(-1.25+2.25*cosinc*cosinc));

   return 0.5*(R_perp*R_perp + R_par*R_par);

}
```

Note that it would be possible to calculate one function from the other, but the implementation requires that the functions are kept distinct.

4) a function for the reflectivity of the internal face of the glass, for rays that come from outer space (the glass). Supposing that the two glass faces are identical, we can use the function defined at point 3; we simply change the name, calling it `ReflectGlassInternOut`. Here we are neglecting the little difference between the refraction index of air and of vacuum.

5) A function for the reflectivity of the internal face of the glass, for rays that come from inner space (the vacuum between the glass and the absorber). As before, we copy the function already defined at point 2, changing its name to `ReflectGlassInternIn.`

6) Reflectivity function of the absorber: we can use a fictitious model similar to the one used for the mirror, with a normal reflectivity of 0.05:

```
double ReflectAbsorber (double cosinc, double lambda) {

   return 1-0.95*cosinc;
}
```

So, the 6 reflectivity functions are defined.

One must also define the 3 refractive indices, for the 3 media that the ray of light can travel through: air, glass, and vacuum. For simplicity, we use here a constant index of refraction for all the media:

1) air:

```
double RefIndex_air (double lambda) {
   return 1.0003;
}
```

2) glass:

```
double RefIndex_glass (double lambda) {
   return 1.5;
}
```

3) vacuum:

```
double RefIndex_vacuum (double lambda) {
   return 1;
}
```

The three media also require a function of absorption of light. In this simple model, we consider null absorption by air and vacuum, and an exponential decay law for the glass, with an absorption of 0.3% of the incident light for a glass thickness of one millimeter.

1) Air:

```
double Absorption_air(double length, double lambda) {
   return 1;
}
```

2) Glass:

```
double Absorption_glass(double length, double lambda) {
   return exp(-3*length);
}
```

3) Vacuum:

```
double Absorption_vacuum(double length, double lambda) {
   return 1;
}
```

So for the system considered here one must define 12 optical functions.

All the functions are independent: the refraction index has nothing to do with reflectivity, nor with the absorption. This feature allows to simulate thin films, and to use wavelength-mean properties. As an example, a thin antireflex layer on the glass changes the properties of reflection of the glass, that cannot be calculated from the simple index of refraction; and it is not possible to simulate the anti-reflex as another concentric layer, because it is too thin and wave optics would be needed. So, in the case of an anti-reflex layer the glass must be described as a glass with the usual index of reflection, but with a law of reflectivity on the surface not calculable from the index of refraction, and defined otherwise.

Of course, the definition of the function is quite laborious, and it is better to do it once and for all and save the functions in a header file. A file of this kind is `opt_fun_definition.h`, in which the optical functions are already defined; one can directly change the functions in this file to save work.

Once the functions have been defined, pointers to functions must be assigned to them, to pass them to the constructor of the `opt_simulation` class. Suitable pointer types are defined in the library. The type `reflect_s` points to a function of reflectivity, the type `refrac_index_function` points to a refractive index function, and the type `absorption_function` points to an absorption function. The assignment must be made in this way:

- a pointer of the type `reflect_s` must be associated to the fucntion of reflectivity of the mirror:

```
reflect_s ReflectMir = &MirrorReflectivity;
```

- an array of 2*NumSurface-1 pointers of the type reflect_s must be declared and associated with the reflective functions of the concentric surfaces, in order from the outside to the inside: in this case, the first function (index 0) is the reflectivity of the external face of the glass for outer rays, the second (index 1) is the reflectivity of the external face for inner rays, followed by the functions for the internal face in the same order; the last function is the reflectivity of the absorber:

```
reflect_s ReflectSurfaces[5];

ReflectSurfaces[0] = &ReflectGlassExternOut;
ReflectSurfaces[1] = &ReflectGlassExternIn;
ReflectSurfaces[2] = &ReflectGlassInternOut;
ReflectSurfaces[3] = &ReflectGlassInternIn;
ReflectSurfaces[4] = &ReflectAbsorber;
```

- an array of NumSurface pointers of the type refrac_index_function must be assigned to the refraction index functions, in order from the outermost to the inmost medium (in this case: air, glass, vacuum):

```
refrac_index_function RefInd[3];

RefInd[0]=&RefIndex_air;
RefInd[1]=&RefIndex_glass;
RefInd[2]=&RefIndex_vacuum;
```

- an array of NumSurface pointers of the type absorption_function must be assigned to the function of absorption, in the same order as above:

```
absorption_function Absorption[3];

Absorption[0]=&Absorption_air;
Absorption[1]=&Absorption_glass;
Absorption[2]=&Absorption_vacuum;
```

The definition and assignment of the optical functions is concluded. When constructing the object opt_simulation, the pointers must be passed to the constructor, after the geometrical parameters, in the following order: reflectivity of mirror, array of the reflectivity of the surfaces, array of the refraction indices, array of the absorption functions. This is the meaning of the command

```
opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
               RadiiReceiver, ReflectMir, ReflectSurfaces, RefInd,
               Absorption);
```

in the example of Section 3.

In that example, all the work shown here was avoided: the definition of the functions was in the opt_fun_definition.h file, and the assignation was hidden in the OPT_FUN_INIT macro, also defined in opt_fun_definition.h. This header file can always be used to save work, even forgetting all the machinery of the function pointers, since it is plausible that the optical properties will not be changed too often. One can proceed in this way:

- modify the optical function directly in `opt_fun_definition.h`, to reproduce the system; the comments in the file indicate clearly the role of each function, for the classical configuration of a glass-covered receiver. If the desired configuration is different, functions can be added or deleted; in this case, the size of the arrays of pointers and the assignment to the pointers must also be changed consistently.
- include the header file in the program, *after* the `SimulTrough.h` file (since some of the definitions and functions of the library are used in `opt_fun_definition.h`);
- before creating the `opt_simulation` object that will perform the simulation, call the `OPT_FUN_INIT` macro;
- create the `opt_simulation` object passing to the constructor the variables `ReflectMir, ReflectSurfaces, RefInd, Absorption` (respectively at the position 4, 5, 6 and 7 of the argument list) as optical function pointers, since these are the names used by the macro.

This procedure has already been shown in the example in Section 3; now the meaning of the few lines used to initialize the optical properties should be clear.

Now the complete header file, as it is given initially with the library, is shown. We already built all its parts in the previous examples of optical functions, in this section. The definition and the assignment of the pointers are in the macro definition.

```
// header file opt_fun_definition.h

#ifndef OPT_FUN_DEFINITION
#define OPT_FUN_DEFINITION


// DEFINITIONS OF THE REFLECTIVITY FUNCTIONS //

// the reflectivity can depend on the angle of incidence and
// on the wavelength of the light. In the following functions,
// the variable cosinc is the cosine of the incidence angle
// on the surface, and lambda is the wavelength of the
// incident light, in Angstrom

// MirrorReflectivity: reflectivity function of the mirror

double MirrorReflectivity (double cosinc, double lambda) {
   return 1-0.1*cosinc;
}


// ReflectGlassExternOut: reflectivity of the external face of
// the glass, when the light comes from the outside (from the air)

double ReflectGlassExternOut (double cosinc, double lambda) {

   double R_perp = (cosinc-sqrt(1.25+cosinc*cosinc))/(cosinc+
         sqrt(1.25+cosinc*cosinc));

   double R_par = (2.25*cosinc-
         sqrt(1.25+cosinc*cosinc))/(2.25*cosinc+
         sqrt(1.25+cosinc*cosinc));

   return 0.5*(R_perp*R_perp+R_par*R_par);
}


// ReflectGlassExternIn: reflectivity of the external face of
// the glass, when the light comes from the inside (from the glass)

double ReflectGlassExternIn (double cosinc, double lambda) {

   if ( cosinc*cosinc <= 0.5556)  return 1; // total reflection

   double R_perp = (cosinc-sqrt(-0.5556+cosinc*cosinc))/(cosinc
         +sqrt(-0.5556+cosinc*cosinc));

   double R_par = (cosinc-1.5*sqrt(-1.25
         +2.25*cosinc*cosinc))/(cosinc+
         1.5*sqrt(-1.25+2.25*cosinc*cosinc));

   return 0.5*(R_perp*R_perp + R_par*R_par);
}


// ReflectGlassInternOut: reflectivity of the internal face of
// the glass, when the light comes from the “outside”
// (from the glass)

double ReflectGlassInternOut (double cosinc, double lambda) {

   if ( cosinc*cosinc <= 0.5556)  return 1; // total reflection

   double R_perp = (cosinc-sqrt(-0.5556+cosinc*cosinc))/(cosinc
         +sqrt(-0.5556+cosinc*cosinc));

   double R_par = (cosinc-1.5*sqrt(-1.25
         +2.25*cosinc*cosinc))/(cosinc+
         1.5*sqrt(-1.25+2.25*cosinc*cosinc));

   return 0.5*(R_perp*R_perp + R_par*R_par);
}


// ReflectGlassInternIn: reflectivity of the internal face of
// the glass, when the light comes from the “inside”
// (from the vacuum)

double ReflectGlassInternIn (double cosinc, double lambda) {

   double R_perp = (cosinc-sqrt(1.25+cosinc*cosinc))/(cosinc+
         sqrt(1.25+cosinc*cosinc));

   double R_par = (2.25*cosinc-sqrt(1.25
         +cosinc*cosinc))/(2.25*cosinc+
         sqrt(1.25+cosinc*cosinc));
   return 0.5*(R_perp*R_perp+R_par*R_par);
}



// ReflectAbsorber: reflectivity of the absorbing surface

double ReflectAbsorber (double cosinc, double lambda) {
   return 1-0.95*cosinc;
}



// DEFINITIONS OF REFRACTIVE INDEX FUNCTIONS //

// in the refractive index functions, the only dependence is
// from the wavelength of the light. The parameter lambda
// is the wavelength in Angstrom

// RefIndex_air: refractive index of the outer space (the air)
double RefIndex_air (double lambda) { return 1.0003; }

// RefIndex_glass: refractive index of the glass
double RefIndex_glass (double lambda) { return 1.5; }

// RefIndex_vacuum: refractive index of the vacuum
double RefIndex_vacuum(double lambda) { return 1; }



// DEFINITIONS OF THE ABSORPTION FUNCTIONS //

// the absorption can depend on the distance travelled by the light
// in the medium and on the wavelength of the light
// (respectively called length and lambda in the
// following functions).
// The function must return the ratio between the final energy
// of the ray after passing in the medium and the initial energy of
// the ray ( and _not_ the fraction of energy absorbed).


// Absorption_air: absorption function of the air
double Absorption_air(double length, double lambda) { return 1; }

// Absorption_glass: absorption function of the glass
double Absorption_glass(double length, double lambda)
   { return exp(-3*length); }

// Absorption_vacuum: absorption function of the vacuum
double Absorption_vacuum(double length, double lambda)
   { return 1; }


// definition of OPT_FUN_INIT, the macro that assigns the
// optical functions to the pointers

// the types reflect_s and absorption_function are pointers
// to function double(double,double)
// the type refrac_index_function is a pointer to
// a function double(double)
// they are defined in SimulTrough.h

   // the following macro (OPT_FUN_INIT) defines:

      // reflectivity of mirror pointer (ReflectMir)

      // reflectivity of concentric surfaces (absorber included)
      // pointers: array of 2*NumSurface-1 pointers
      // (ReflectSurfaces)

      // refractive indices pointers: array of NumSurface pointers
      // (RefInd)

      // absorption functions pointers: array of NumSurface
      // pointers (Absorption)



#define OPT_FUN_INIT \
      \
      reflect_s ReflectMir = &MirrorReflectivity; \
      \
      \
      reflect_s ReflectSurfaces[5]; \
      ReflectSurfaces[0] = &ReflectGlassExternOut;\
      ReflectSurfaces[1] = &ReflectGlassExternIn;\
      ReflectSurfaces[2] = &ReflectGlassInternOut;\
      ReflectSurfaces[3] = &ReflectGlassInternIn;\
      ReflectSurfaces[4] = &ReflectAbsorber;\
      \
      \
      refrac_index_function RefInd[3];\
      RefInd[0]=&RefIndex_air;\
      RefInd[1]=&RefIndex_glass;\
      RefInd[2]=&RefIndex_vacuum;\
      \
      \
      absorption_function Absorption[3];\
      Absorption[0]=&Absorption_air;\
      Absorption[1]=&Absorption_glass;\
      Absorption[2]=&Absorption_vacuum


#endif

// end of file
```

The `opt_fun_definition.h` file can be used in many applications, except when the optical properties must be changed while the program is running: in this case, one must define all the optical functions involved in the simulation and then work directly with the pointers, passing them to the class when needed. In Section 9, the way to do this will be shown.

5. Initialization of the simulation parameters

The first thing to do to perform a simulation is creating the system to simulate. In the program, the system is represented by an object of the class `opt_simulation`; in the example of Section 3, this object was called `simul`. The class contains the parameters of the collectors, and all the functions that perform the simulation and read the results, and also functions to change the configuration. Here we show in detail the constructor of the class, and the meaning of the various parameters. The constructor is declared in this way:

```
opt_simulation::opt_simulation(

                    // geometric parameters
                    double Focus,
                    double MirrorWidth,
                    double FreeSpace,
                    int NumSurface,
                    double* RadiiReceiver,

                    // optical functions
                    reflect_s ReflectMirror,
                    reflect_s* ReflectSurf,
                    refrac_index_function* RefracIndex,
                    absorption_function* AbsorptionFunctions,

                    // simulation technical parameters
                    int nAnnuli = 10,
                    int nSector = 72,
                    int nStep = 200,
                    int nLambda = 1,
                    double LimitEnergy = 1e-5,
                    int NRicLim = 30,
                    double Offset = 1e-5,

                    // errori
                    double Defocalisation_x = 0,
                    double Defocalisation_y = 0,
                    double GaussTrasv = 0,
                    double GaussLong = 0,
                    int NumberHarm = 0,
                    double* CoeffHarm = 0    )
```

It is possibile to define more than an `opt_simulation` object, and the different objects will be completely independent; but this is not very useful, since the parameters of a system can be modified instead of creating a new object.

The constructor has 9 parameters that must be given in all cases, and 13 optional parameters. The meaning of the first 9 parameters, that we already used in Section 3, is the following:

Geometric parameters of the system:

`double Focus`: focal length of the mirror;
`double MirrorWidth`: half-width of the collector;

`double FreeSpace`: half-width of the possible free space at the vertex of the mirror;

`int NumSurface`: number of concentric cylindrical surfaces around the focal line, at which the light can be reflected, refracted, or absorbed; i. e., 3 for the classical glass-covered receiver (absorber, internal face of the glass, external face of the glass);

`double* RadiiReceiver`: array that contains the radii of the concentric surfaces, from the outermost to the inmost (the absorber is the last one).

Optical functions (discussed in the previous section):

`reflect_s ReflectMirror`: pointer to the reflectivity function of the mirror;

`reflect_s* ReflectSurf`: array of pointers to the reflectivity functions of the concentrical surfaces, from the outermost to the inmost, and with 2 elements (ray from outside / ray from inside) for each surface except for the absorber, that requires only one function (ray from outside);

`refrac_index_function* RefracIndex`: array of pointers to the index refraction functions of the various media, from the outermost (air) to the inmost (vacuum);

`absorption_function* AbsorptionFunctions`: array of pointers to the absoption functions, from the outermost (air) to the inmost (vacuum).

Following the method explained at the end of Section 4 to initialize the optical functions, with the macro `OPT_FUN_INIT,` these four variables will always be called, respectively, `ReflectMir`, `ReflectSurfaces`, `RefInd`, `Absorption`.

The initialization procedure is shown in the beginning of the previous example:

```
#include"(path)\SimulTrough.h"
#include"(path)\opt_fun_definition.h"


int main() {

OPT_FUN_INIT;

int NumSurface = 3;

double Focus = 1.8,
   MirrorWidth = 3.0,
   FreeSpace = 0.05;

double RadiiReceiver[] = {0.065, 0.062, 0.035};

opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
   RadiiReceiver, ReflectMir, ReflectSurfaces, RefInd, Absorption);

   (…)
```

where the geometrical parameters are first assigned to variables (only for clearness), and then the variables are passed to the constructor. After these program lines, one has an object that represents the system, on which it is possible to work using the member functions of the class; among them, there is the function simulate that performs the optical simulation.

The meaning of the parameters, in the case of a classical glass-covered receiver, is shown in Figure 1.

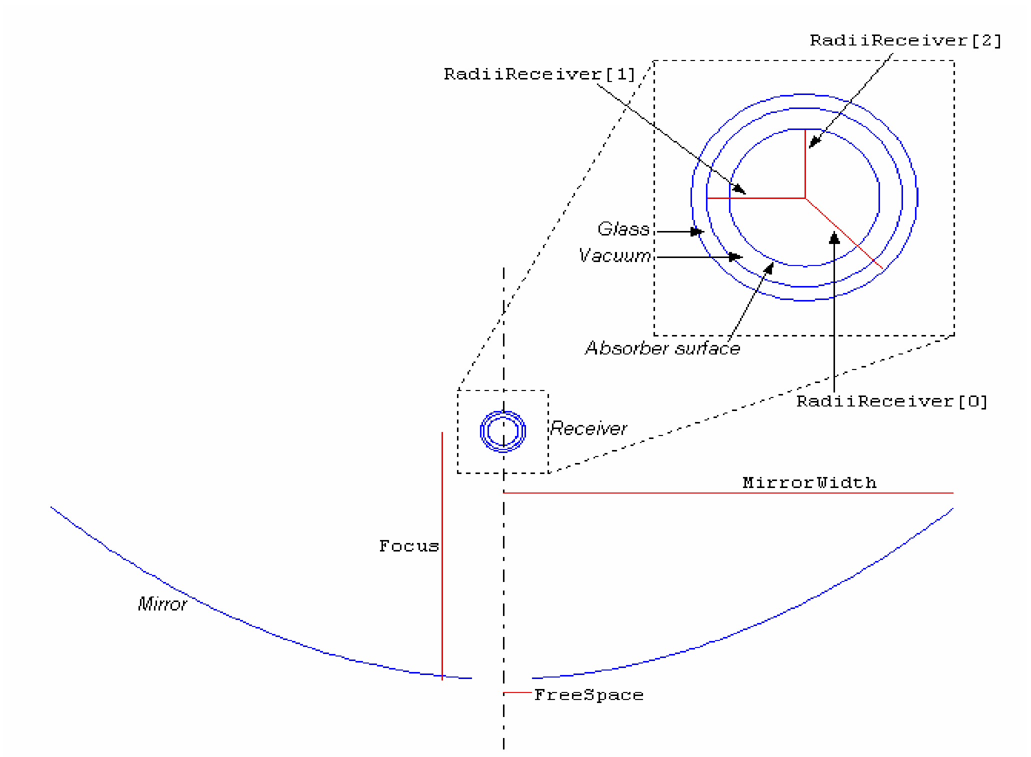


Figure 1: Geometric parameters of the system, in the case of a glass-covered evacuated tube as a receiver.

The 13 optional parameters are set to default values, shown in the declaration of the constructor. They are divided in two groups: the first 7 are technical parameters of the simulation, that set the number of rays shot, the number of angular sectors in which the circumference of the receiver is divided, the accuracy of the wavelength analysis, and numerical limits on the recursions and on the energy of rays. The last 6 parameters define the defects of the collector: the default is a perfect collector. All these parameters can be changed later. They will be discussed in Sections 10 and 11.

6. Performing the simulation

Once an object of the class opt_simulation has been created, the optical simulation is performed with the function simulate of the class, declared in this way:

```
void opt_simulation::simulate(double x, double y, double z,
   double shadow_sx = 0, double shadow_dx = 0);
```

The last two optional parameters are for the shadowing and will be explained later; for now, they will be omitted, which corresponds to complete exposition to the sun. The first three parameters of the function are the three components *x*, *y* and *z* of the vector that points towards the centre of the sun. Its norm is arbitrary, but in order to avoid numerical difficulties it is preferable not to choose very long or very short vectors; a norm of the order of unity is recommended. In the reference system used here**,** the plane *xy* contains the parabolic section, with the focus and the vertex at y=0 (the parabolic profile equation is $y=x^2/4f$, where f is the focal distance), and the z axis lies on the vertex line of the parabolic mirror. The coordinate system is shown in Figure 2.

As an example, if the collimation is perfect (no tracking error), and the sun is perfectly orthogonal to the collection plane, the vector is (x,y,z) = (0,1,0). The tracking error can be introduced by rotating the vector in the *xy* plane: a tracking error of an angle a is represented by the vector (x,y,z) = (-sin a, cos a, 0). But of course, even without tracking errors, the sun is usually not orthogonal to the collector (orthogonality happens at most twice a day); the inclination γ of the sun with respect to the collector plane, with no tracking error, is represented rotating the vector in the plane *yz*. The vector is (x,y,z) = (0, cos γ, -sinγ). A generic vector (a,b,c) without null elements represents a situation in which the sun is not orthogonal and a tracking error is present. The situation with y=0 has no practical utility (the rays cannot hit the collector).

To perform the simulation, once we have declared an object `opt_simulation` (called `simul`), the command to perform the simulation is

```
simul.simulate(0,1,0);
```

that simulates the case of perfect orthogonal incidence. It can be added to the code lines shown in Section 5.

The function `simulate` is a member of the object `simul`, and it uses all the properties of the object defined in its construction (see Section 5). Other members of `simul` are variables in which the simulation results are saved, and functions to read them. These aspects will be discussed in the next section.

During the simulation, the displayed output is:

```
Setting solar position ..... OK

x0 = -2.95     OK
x0 = -2.9205   OK
x0 = -2.9891
```

`x0` is the coordinate x from which the solar beam is shot; when `OK` appears, the analysis for that point is finished. The simulation ends when `x0` reaches the end of the collector, at about `MirrorWidth` (with a little variation if there is a tracking error). If the printing of the process is not desired it can be disabled with the function `void opt_simulation::printoff()`, simply inserting the command

```
simul.printoff();
```

before the simulation. The opposite command is

```
simul.printon();
```

that enables again the printing of the process.

In the simulation, the rays are shot on the mirror and followed along all the reflections and refractions due to all the surfaces in the system, until they become negligible. The analysis of the single ray is fully three-dimensional. When a ray hits a refracting surface, two rays are generated: the reflected and the refracted ray, and both of them are followed; a recursive "tree" of secondary rays is built from the original ray shot on the mirror. A maximum number of recursion is set, after which the rays are neglected in all cases, but this number is almost never reached except in some very particular cases (i. e. when total reflection in nonabsorbing media occurs). The energy of all the neglected rays is saved in a variable useful for control. Two examples of the multiple reflections and refractions that occur for a glass-covered receiver are shown in Figure 3.

A solar beam is a set of rays that reproduce the solar angular distribution of energy. The beam is built dividing the sun disk in a number of annuli and placing in each annulus a number of rays regularly spaced, with a random offset with respect to the next annulus. The distribution is built as uniformly as possible in the solar disk; moreover, after positioning the rays, a correction of the energy of the rays is applied, so as to fix the dishomogeneity in the discretisation and set the limb-darkened profile. The same procedure is used to build the halo, with greater spacing. The number of annuli can be set by the user, and the number of rays in a beam is proportional to the square of the number of annuli. The default value of the number of annuli is 10, corresponding to 385 rays for each sun beam.

The limb-darkened distribution used is shown in Figure 4.

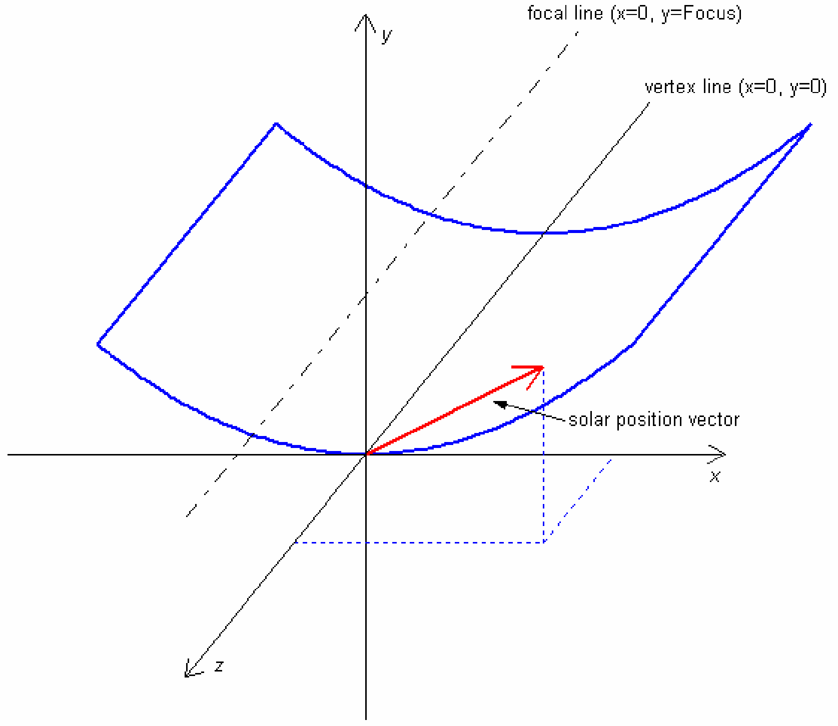


Figure 2: Coordinate system used to pass the the sun position to the `simulate` function.

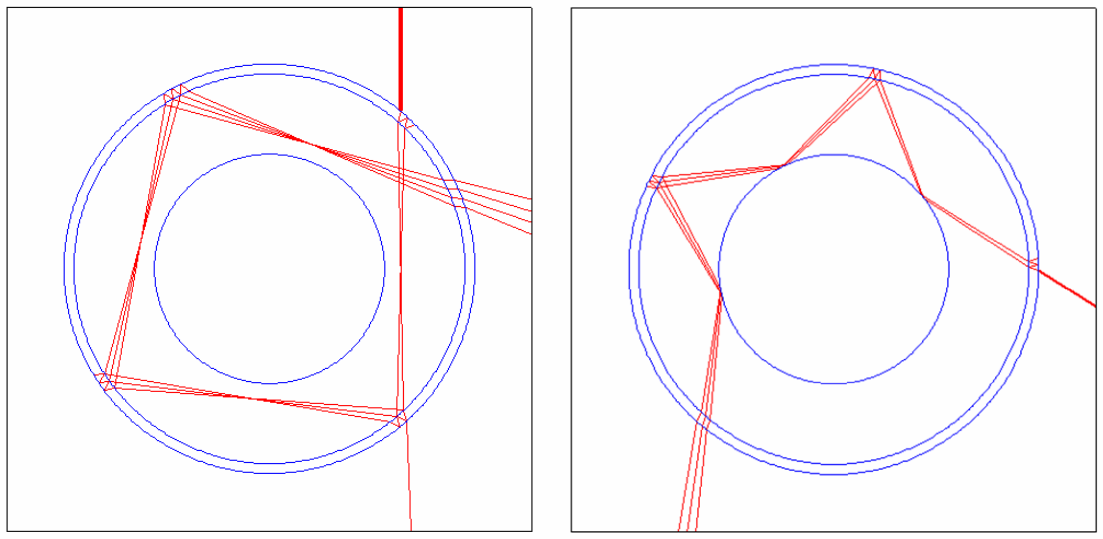


Figure 3: Multiple reflections and refractions of an incident ray (the thick line) on the receiver glass covering and on the absorbing surface (the mirror is below). Left: the ray hits directly the receiver glass from above, and it is scattered. Right: the ray comes from the mirror, with a small tracking error.

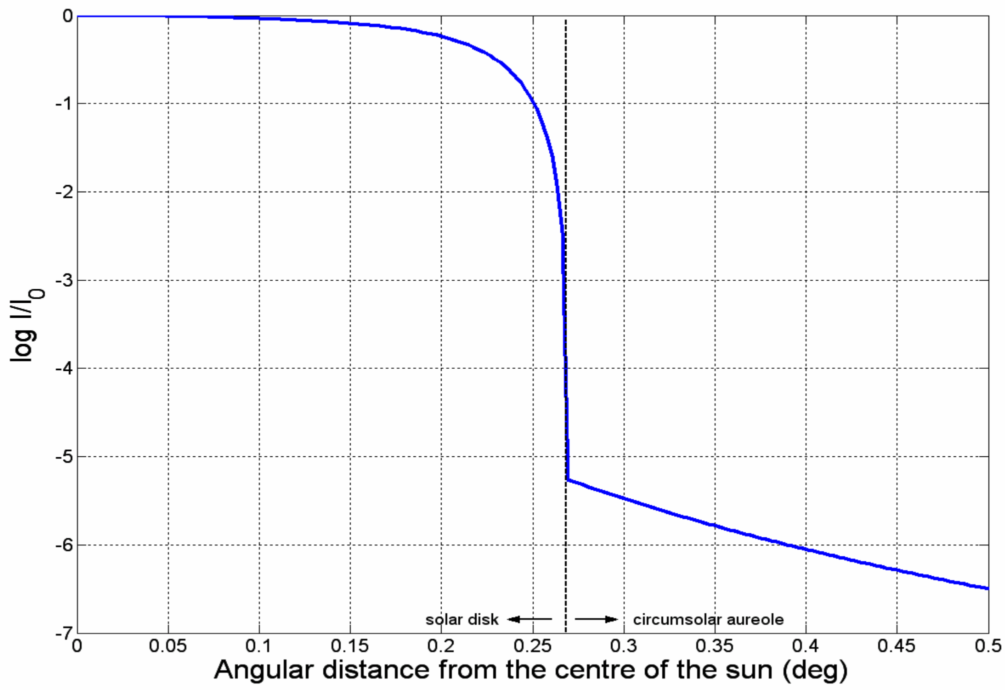


Figure 4: Distribution of the radiation intensity (in logarithmic scale) used to simulate the sun (I_0 = radiation intensity at the centre of the sun).

A number of solar beams is then shot on the collector surface, at regular intervals, from above the receiver (so the receiver shadowing effect is reproduced, and the light that hits directly the receiver is collected). The number of beams shot can be set by the user, but it should be high enough to have a rather uniformly distributed irradiation on the collector (a minimum of 100 beams is recommended). The default value is 200. So, in a default simulation a total of 77000 rays is shot. The time required, for normal incidence and without introducing defects in the system, is less than 10 seconds on a 3 GHz Pentium 4 CPU, 2 MB RAM machine. But the time can increase strongly for high tracking errors or strong inclinations.

If a wavelength analysis is performed, the solar spectrum contained in the library is discretised using a certain number of wavelengths, set by the user. For each wavelength, the associated fraction of energy is computed, and a ray is shot. Since the spectrum included in the library is given as a table of 95 wavelengths, it is useless to set a number of wavelengths higher than this. The default number of wavelengths is 1, meaning that no wavelength analysis is performed; the wavelength of the rays is set to the mean wavelength of the solar energy. If the defined optical functions do not depend on the wavelength – as is the case in almost all the examples shown here – the wavelength data are useless.

The setting of these and other simulation parameters will be discussed in Section 10.

7. Results

The simulation computes the distribution of the radiation on the absorber. In order to do that, the circumference of the absorber is divided into a certain number of equal angular sectors, and when a ray is partially absorbed in a sector the absorbed energy is added to the energy of that sector. To read the energy absorbed in a sector, once the simulation has been performed, there is an apposite function:

```
double opt_simulation::get_absorber(int NumberSector);
```

The parameter `NumberSector` is an integer that gives the number of the sector. The default number of sectors is 72, with an amplitude of 5° each. The sectors are numbered progressively starting from 0, starting from the point of the receiver opposite to the collector and proceeding in counterclockwise direction: with 72 sectors, the two sectors nearest to the collectors are the number 35 and 36, and the half-pipe facing the collector goes from sector 18 to sector 53 included.

As an example, once the simulation has been done, it is possible to print on the screen the energy absorbed between 175° and 180° with the command

```
cout<<simul.get_absorber(35);
```

or storing the distribution in an array called `DistribEnergy` with the lines

```
double DistribEnergy[72];
for (int i=0; i<72; i++) DistribEnergy[i]=simul.get_absorber(i);
```

or saving the distribution in the file `distrib_absorber_ec.dat`, with the lines

```
ofstream of("distrib_absorber_ec.dat");
for (int i=0; i<72; i++) of<<simul.get_absorber(i)<<endl;
```

and using the file to plot the distribution with graphical applications, if desired.

The total direct normal irradiation (DNI) considered is unitary, the values of the energy are partial efficiencies relative to each sector. The total efficiency relative to the DNI is the sum of the energies absorbed in each sector. It can be obtained with the command

```
simul.get_efficiency();
```

that returns a `double` between 0 and 1.

The distribution of the energy absorbed while travelling between the first and the second concentric surface (if they are present) is also saved: this choice is due to the fact that usually this is the space occupied by the glass covering, and the distribution of the energy absorbed in the glass is often required. But this is an “ad hoc” choice that of course result useless if the receiver configuration is different from the classical one, in all the examples. The division in sector is the same used for the absorber, and the function that reads the energy of a sector is similar:

```
double opt_simulation::get_absorb_glass(int NumberSector);
```

Another information that is saved is the energy lost due to neglecting rays when they become negligible, or when there are too many recursions. The sum of all these energies is saved and can be read with the function

```
double opt_simulation::get_rec_error();
```

This is a control function: the result must be much smaller than 1, otherwise the simulation is unreliable. It is *not* an estimation of the error on the efficiency, that is probably greater and is due to other factors (discretisation and numerical approximations).

8. A second example

A slightly more complex example will now be shown. The system considered is the same as in the preceding example: a collector 6 m wide, with a receiver of 7 cm of diameter, covered by a glass of 13 cm of diameter, 3 mm thick. The optical functions are already defined in the file `opt_fun_definition.h`, that is included.

The program performs a simulation with a tracking error of 0.01 rad, saves the distribution of energy absorbed in the file `distrib_absorber_ec.dat`, and also saves the energy absorbed by the glass in the file `distrib_glass_ec.dat`; then the total collection efficiency is printed on the screen. Then a second simulation is performed (turning off the progress output) with the sun at 60° of inclination with respect to the collector plane, with no tracking error; the distribution of the absorbed energy, and the energy absorbed by the glass, are printed on the screen as two lists. Then the control error obtained from `get_rec_error()` is printed.

```
#include<iostream>
#include<fstream>
```

```
#include<cmath>

#include"SimulTrough.h"
#include"opt_fun_definition.h"

using namespace std;

int main() {

// initialization of the optical functions:

   OPT_FUN_INIT;

// definitions of the geometrical parameters:

   int NumSurface = 3;

   double Focus = 1.8,
         MirrorWidth = 3,
         FreeSpace = 0.05;

   double RadiiReceiver[] = {0.065, 0.062, 0.035};

// the object simul is built:

   opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
         RadiiReceiver, ReflectMir, ReflectSurfaces,
         RefInd, Absorption);


         // ***** first simulation *****

   cout<<"---- FIRST SIMULATION ----"<<endl<<endl;

// tracking error of 0.01 rad:

   double ErrorColl = .01;
   simul.simulate( -sin(ErrorColl), cos(ErrorColl), 0 );


// the energy distributions are saved in files:

   ofstream of("distrib_absorber_ec.dat");
   ofstream of2("distrib_glass_ec.dat");

   for (int i=0; i<72; i++) {
         of<<simul.get_absorber(i)<<endl;
         of2<<simul.get_absorb_glass(i)<<endl;
   }


// output the total efficiency

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;
   cout<<"Control error = "<<
         simul.get_rec_error()<<endl<<endl<<endl;


         // ***** second simulation *****
   cout<<"---- SECOND SIMULATION ----"<<endl<<endl;

// printing of the simulation progress is disabled

   simul.printoff();

// inclination of the sun: PI/3 rad, no tracking error

   double Incl = 3.141592/3.0;
   simul.simulate( 0, cos(Incl), -sin(Incl));


// the distributions of absorbed energy are listed

   cout<<"Energy distribution on the absorber:"<<endl;
   for (int i=0; i<72; i++) cout<<"Energy absorbed between "<<5*i
         <<"° and "<<5*(i+1)<<"° = "<<simul.get_absorber(i)<<endl;

   cout<<endl;

   cout<<"Distribution of the energy absorbed by the glass:"<<endl;
   for (int i=0; i<72; i++) cout<<"Energy absorbed between "<<5*i
         <<"° and "<<5*(i+1)<<"° = "<<
         simul.get_absorb_glass(i)<<endl;

// calculation of the total energy absorbed by the glass

   double e_glass=0;
   for (int i=0; i<72; i++) e_glass += simul.get_absorb_glass(i);


// the efficiency, the fraction of energy absorbed by the glass,
// and the control error are printed:

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;
   cout<<"Energy absorbed by the glass = "<<e_glass<<endl;
   cout<<"Control error = "<<
         simul.get_rec_error()<<endl<<endl<<endl;

}
```

The output of the program will be:

```
---- FIRST SIMULATION ----

Setting solar position ..... OK

x0 = -3.05189  OK
x0 = -3.02189  OK
x0 = -2.99189  OK
```

(...)

```
x0 = 2.94811   OK

Efficiency = 0.576216
Control error = 9.91601e-005


---- SECOND SIMULATION ----

Energy distribution on the absorber:
Energy absorbed between 0° and 5° = 0.000106122
Energy absorbed between 5° and 10° = 0.000111513
```

(...)

```
Energy absorbed between 355° and 360° = 0.000104573

Distribution of the energy absorbed by the glass:
Energy absorbed between 0° and 5° = 6.58859e-006
Energy absorbed between 5° and 10° = 6.69652e-006
```

(...)

```
Energy absorbed between 355° and 360° = 6.43763e-006

Efficiency = 0.192857
Energy absorbed by the glass = 0.00788523
Control error = 0.000310983
```

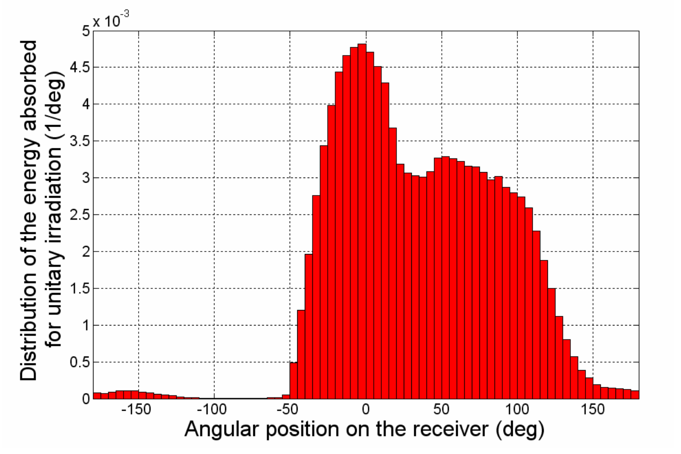
From the first simulation, we see that the efficiency with a tracking error of 0.01 rad (0.57°) is 57.6%, with the optical properties of Section 4 (the efficiency for perfect normal incidence is 77.4%). The two files saved in this simulation can be used to plot the distribution of energy absorbed by the absorber and by the glass. From the second simulation, we see that the control error (0.03%) is negligible, as expected, even for a strong inclination. The efficiency is strongly reduced due to the inclination of 60°, and it is only 19.3%; it must be stressed that this efficiency already includes the cosine factor (in this case 0.5), since the simulation is carried out with a unitary DNI. The 0.8% of the energy is absorbed by the glass.

We now can use the files saved in the first simulation to plot the energy distribution with some graphical program. The distributions obtained from the first simulation in the above example are shown in Figure 5.

9. Changing geometrical and optical parameters

All the geometrical and optical parameters of an object can be read and modified. The functions that reads the geometrical parameters are, with obvious meaning and use:

```
double opt_simulation::get_Focus();
double opt_simulation::get_MirrorWidth();
double opt_simulation::get_NumSurface();
double opt_simulation::get_RadiiReceiver(int number_surface);
double opt_simulation::get_FreeSpace();
```



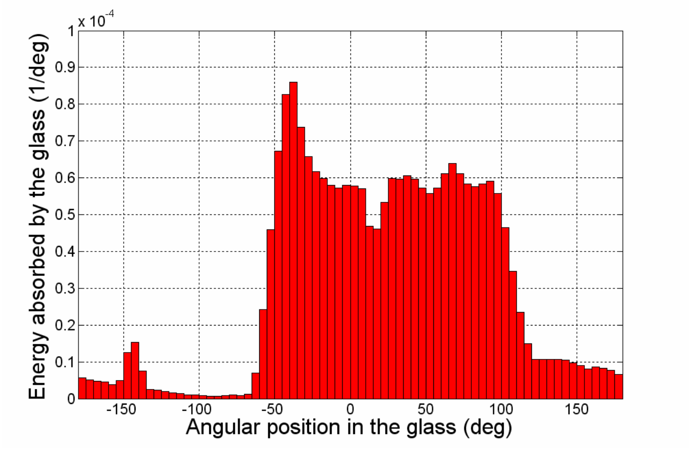


Figure 5: Distribution of the energy absorbed by the receiver (left) and by the glass (right), with a tracking error of 0.01 rad (first simulation). The distribution of the energy in the glass shows some irregularities, probably due to discretisation errors (the absorbed energy is very small, and the simulation is not very accurate, lasting only 10 seconds); the distribution would probably be smoother with a higher number of steps. However, the precision is sufficient to build a good distribution of the energy absorbed by the receiver surface (left).

The function `get_RadiiReceiver` needs the number that identifies the considered surface. As usual, the surfaces are numbered from the outermost to the inmost, starting from 0.

The geometrical parameters, except the number of surfaces (that will be discussed later), can be modified with the functions

```
void opt_simulation::set_Focus(double newFocus);
void opt_simulation::set_MirrorWidth(double newMirrorWidth);
void opt_simulation::set_RadiiReceiver(int NumberSurface,
   double newRadius);
void opt_simulation::set_FreeSpace(double newFreeSpace);
```

that overwrites the old values (`set_RadiiReceiver` requires the number of the surface whose radius is modified). It should be pointed out that changing a parameter does *not* automatically restart the simulation, and the results previously collected, if any, remains unchanged until a new command `simulate` is given.

There is also the function

```
void opt_simulation::set_geometry(double newFocus,
   double newMirrorWidth, double newFreeSpace,
   double* newRadiiReceiver);
```

that performs all the changes in one command; here, `newRadiiReceiver` must be a vector, with the list of the radii, from the outermost to the inmost.

A more radical change can involve the number of concentric surfaces: it is possible to change it, but new optical functions must be assigned, since their number (and consequently the dimensions of the arrays of function pointers) must change. All this can be done with the function

```
void opt_simulation::set_NumSurface(int newNumSurface,
      double* newRadiiReceiver, reflect_s* list_newReflectSurf,
      refrac_index_function* list_newRefInd,
      absorption_function* list_newAbsorption);
```

where `newNumSurface` is the new number of concentric surfaces, `newRadiiReceiver` is the new array of the radii of the concentric surfaces, `list_newReflectSurf` is the array of pointers to the reflectivity functions, `list_newRefInd` is the array of pointers to the refractive indices, and `list_newAbsorption` is the array of pointers to the absorption functions.

The optical functions can also be changed one by one with the following functions, with obvious meaning

```
   void opt_simulation::set_ReflectMirror(reflect_s newReflectMir);
   void opt_simulation::set_ReflectSurfaces(int NumberFun, reflect_s
                  newReflectSurface);
   void opt_simulation::set_RefracIndex(int NumberFun,
                  refrac_index_function newRefIndex);
   void opt_simulation::set_Absorption(int NumberFun,
                  absorption_function newAbsorption);
```

where `NumberFun` is the index of the array where the function must be changed. It must be stressed that the arguments are all pointers to functions, and not arrays of pointers: the new optical functions are placed in the old arrays.

In practical use, if many changes must be applied to the system, it may be convenient to modify the file `opt_fun_definition.h` and restart the simulation, instead of acting on the optical functions from the program.

Now, some examples of the functions described. At the end of the example of Section 8, we can decide to change the focal length to 2.5 m, and the radius of the receiver to 4 cm instead of 3.5, and then perform a new simulation with the new data: this can be made with the lines

```
   simul.set_Focus(2.5);
   simul.set_RadiiReceiver(2, 0.04);
   simul.simulate( sin(ErrorColl), -cos(ErrorColl), 0 );
```

or also

```
   double newRadiiReceiver[] = {0.065, 0.062, 0.04};
   simul.set_geometry(2.5, MirrorWidth, FreeSpace, newRadiiReceiver);
   simul.simulate( sin(ErrorColl), -cos(ErrorColl), 0);
```

In the first case (recommended) we change the focal length with the command `set_Focus`, and the radius of the third surface (the count starts from 0!), that is the absorber. In the second case, we define a new array of radii and we pass it to the command `set_geometry`, together with the new value of the focal length and the old values for the other parameters. The second method is more cumbersome and requires to define a new array.

Let us see yet another example. Suppose that, collecting experimental data or making simulation, we have defined a new reflectivity function for a dirty glass surface: `double reflect_dirty(double cosinc, double lambda)`, and we desire to replace the old reflectivity function for the external face of the glass in order to simulate the effect of exposition to dust and rain, and make a comparison with the clean glass. Once the results for the clean glass have been saved, if the function `reflect_dirty` is visible to the program we can change the function of reflectivity with the commands:

```
reflect_s new_refl = &reflect_dirty;
simul.set_ReflectSurfaces(0, new_refl);
```

In these lines we define a pointer of the type `reflect_s` (a pointer to a reflectivity function), we associate it with our new reflectivity function (`reflect_dirty`), and we pass it to the array of the pointers to the reflectivity functions, in the first position. (Then we should also change the reflectivity for rays coming from the glass, at the position 1 of the array).

10. Technical parameters of the simulation

In the constructor of the object `opt_simulation`, shown in Section 5, there are also other parameters, set to default values if omitted. Now we will discuss their meaning and use.

`int nAnnuli` (default 10)
The solar disk is divided into circular annuli, to build the distribution of rays; in each circular annulus a uniformly spaced distribution of rays is defined, with the number of rays proportional to the radius and with a spacing comparable to the width of the annulus. The energy of the rays is then modulated to correct the discretisation errors and to follow the limb-darkened model. A fraction of rays (the 10%) is used for the halo, where the spacing is larger.

The quantity `nAnnuli` is the number of annuli in which the solar disk (plus the halo) is divided. The higher is the number, the more uniform and well-reproduced is the solar beam. The number of rays of each solar beam is proportional to the square of `nAnnuli`; so, for example, if the number of annuli is increased from 10 to 20, the simulation time is multiplied by a factor 4; if it is increased to 50, the factor is 25.

`int nSector` (default 72)
It is the number of angular sectors in which the circumference of the receiver is divided, to build the distribution of the absorbed energy. Increasing this number, the resolution is higher, but if the number of rays is not high enough there can be purely numerical fluctuations due to the discrete analysis. The number of sectors should be adequate to the number of rays shot. By default, the circumference is divided into 72 sectors of 5° of amplitude. Neither the total efficiency precision nor the simulation time are affected by changing this parameter; only the way the results are saved is changed.

It is worthwhile to remark that, changing the number of sectors when a simulation has been done, the old results are lost. The number of sectors must always be set *before* performing the simulation, i.e., before the command `simulate`.

`int nStep` (default 200)

It is the number of regular intervals in which the collector width is divided; at the edge of each of these intervals, a solar beam is shot. Increasing `nStep` the radiation distribution is more homogeneous and the precision increases. The simulation time grows linearly with `nStep`.

`int nLambda` (default 1)

When the wavelength dependence of the optical properties is considered, the solar wavelength spectrum is discretised using a certain number `nLambda` of wavelength (not regularly spaced), and for each ray of the solar beam, a number `nLambda` of rays with defined wavelength and with the corresponding energy are shot, summing the results at the end. In the default case (`nLambda` = 1) the wavelength dependence is not considered; if some of the functions are wavelength dependent, the wavelength considered is the mean for the solar spectrum. Since the spectrum in the library has 95 subdivisions, it is useless to give a number `nLambda` higher than 95.

The simulation time grows linearly with `nLambda`.

`double LimitEnergy` (default 1e-5)

When a ray has an energy smaller than this quantity, it is considered lost. The simulation time increases when `LimitEnergy` is decreased.

`int NRicLim` (default 30)

A ray is reflected and refracted generating new rays, treated with a recursive procedure. To avoid infinite recursions, a limit `NRicLim` to the recursion level is set. Usually, this limit has no effect on the simulation, except in very special cases (for example, rays imprisoned by total reflection in a nonabsorbing medium).

`double Offset` (default 1e-5)

It is the initial offset of the rays starting from a surface, to avoid computing a spurious intersection with the starting surface is found. It must be less than the minimum distance between two surfaces of the system. It has no effects on the simulation precision and time.

If the desired parameters are different from the default parameters, one can define the new parameters directly when building the object, passing values that overwrite the default ones. As an example, with the command

```
opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
   RadiiReceiver,ReflectMir, ReflectSurfaces, RefInd, Absorption,
   20, 90, 500, 10);
```

(where all the variables are defined as in the previous two examples) an object `simul` is created, with the sun divided in 20 annuli, the results are saved in 90 angular sectors of 4°, the simulation is made shooting 500 solar beams and 10 wavelengths for each ray are considered.

All these parameters can also be changed for an existing object, with apposite functions. Some of them are important for the user and will probably be changed frequently (`nAnnuli, nSector, nStep, nLambda`), while others probably will not

(`NRicLim, Offset`). However, functions that read and modify all the parameters are defined in any case. The functions to read the parameters are, with obvious meaning,

```
int opt_simulation::get_nAnnuli();
int opt_simulation::get_nSector();
int opt_simulation::get_nStep();
double opt_simulation::get_LimitEnergy();
int opt_simulation::get_NRicLim();
double opt_simulation::get_Offset();
int opt_simulation::get_nLambda();
```

and there is also the function

```
int opt_simulation::get_nSolarRay();
```

that returns the total number of rays of a solar beam. The functions to modify the parameters are

```
void opt_simulation::set_nAnnuli(int new_nAnnuli);
void opt_simulation::set_nSector(int new_nSector);
void opt_simulation::set_nStep(int new_nStep);
void opt_simulation::set_LimitEnergy(double new_LimitEnergy);
void opt_simulation::set_NRicLim(int new_NRicLim);
void opt_simulation::set_Offset(int new_Offset);
void opt_simulation::set_nLambda(int new_nLambda);
```

Moreover, a function to modify all the technical parameters at once is defined:

```
void opt_simulation::set_param_simulation(int nAnnuli = 10,
               int nSector = 72,
               int nStep = 200,
               int nLambda = 1,
               double LimitEnergy = 1e-5,
               int NRicLim = 30,
               double Offset = 1e-5);
```

Remark: using this function, if some of the last values are omitted, they do not remain unchanged, but they return to the default values, as it can be seen from the declaration.

11. Simulation of defects of the system

It is possible to simulate systems with defects, as an out-of-focus receiver or deformations of the mirror. The following defects can be simulated.

Defocalisation:
The centre of the receiver is not in the focus. The error can be introduced with the function

```
void opt_simulation::set_ErrorDefoc(double ErrorDefoc_x,
         double ErrorDefoc_y);
```

`ErrorDefoc_x` and `ErrorDefoc_y` are the displacements with respect to the correct position, in the directions *x* and *y*, respectively. As an example, if the receiver is a centimeter under the focus (towards the collector) and two centimeters to the right, the displacements are `ErrorDefoc_x = 0.02`, `ErrorDefoc_y = -0.01`. To introduce this defect in the system described by the object `simul`, the command is

```
simul.set_ErrorDefoc( 0.02,-0.01 );
```

Transverse random error on the mirror inclination:

The inclination of the surface of the mirror, in the parabolic section, is changed of a random quantity with a Gaussian distribution. To introduce this error, with standard deviation `sigma`, there is the function

```
void opt_simul::set_ErrorGaussTrasv(double sigma);
```

Longitudinal random error on the mirror inclination:

The inclination of the surface of the mirror, in the direction parallel to the focal line, is changed of a random quantity with a Gaussian distribution. To introduce this error, with standard deviation `sigma`, there is the function

```
void opt_simul::set_ErrorGaussLong(double sigma);
```

The two random errors just seen can describe the quality of construction of the mirror, or damages and deformations on a small scale. They can be different to each other, since there can be directional effects.

Transverse large-scale deformations:

They are the large-scale deformations of the parabolic section, that can be caused by the supports, thermal stress or gravity effects, or also by a non-perfect building of the mirror.

The difference between the real inclination and the theoretical one must be given as a Fourier inclination in the cosine, with a certain number *N* of harmonics:

$$\Delta\theta = \sum_{j=0}^{N-1} a_j \cos j\pi\zeta, \qquad \zeta = \frac{x+L}{2L}.$$

ζ is a variable between 0 and 1, and it is the relative position on the collector: $\zeta=0$ when $x=-L$, $\zeta=1$ when $x=L$. The cosine paramenter varies from 0 (when $x=-L$) to $j\pi$ (when $x=L$). The quantity $\Delta\theta$ is the angle of deviation from the correct inclination, in counterclockwise direction, at the relative point ζ. The use of the Fourier harmonics is chosen because many typical deformations can be approximated well by a very limited number of harmonics, as it is shown in the example below.

To introduce the error, one must give the number *N* of harmonics, and an array of *N* `double`, the coefficients a_j. The functions

```
void opt_simulation::set_ErrorTrasv(int NumberHarm,
   double* CoeffHarm);
```

receives the number N (`NumberHarm`) and the array of the a_j (`CoeffHarm`).

As an example, we can build a "double-focus" deformation, in which each of the two halves of the mirror has a curvature that is slightly greater than the correct one, creating (approximately) two secondary focuses which are slightly closer to the mirror than the correct focus. A profile that reproduces this deformation quite well, shown in Figure 6, is proportional to $-\sin(3\pi\zeta)$; the error on the inclination, for small angles (when $\tan\theta$ can be approximated by θ), will be proportional to $-\cos(3\pi\zeta)$. Supposing that the maximum error on the inclination is 0.01 rad, the deviation $\Delta\theta$ will be $-0.01\cos(3\pi\zeta)$. So, we need four harmonics: a_0, a_1 and a_2 will be 0, and $a_3 = -0.01$. We have N=4, a={0,0,0,–0.01}. In a program, we could introduce this deformation with the commands

```
double c_harm[] = {0,0,0,-0.01};
simul.set_ErrorTrasv(4,c_harm);
```

where we first define the array of the harmonic coefficients and then we pass it to the function `set_ErrorTrasv`, specifying its size.

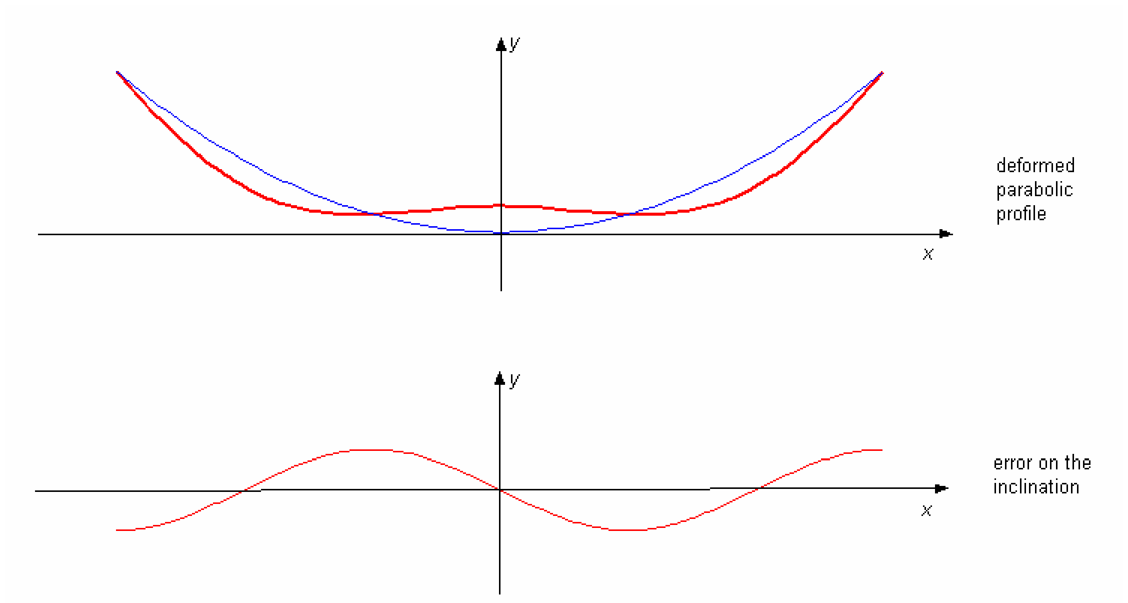


Figure 6: Example of a "double-focus" deformation. Above: the deformed parabolic profile (thick red line) compared with the correct profile (the deformation is exaggerated). Below: the error on the mirror inclination, proportional to $-\cos(3\pi\zeta)$.

A function to introduce all the errors at the same time is also defined:

```
void opt_simulation::set_Errors(double ErrorDefoc_x,
   double ErrorDefoc_y, double ErrorGaussTrasv,
   double ErrorGaussLong, int NumberHarm, double* CoeffHarm);
```

It is also possible to define directly the system defects when the class is created, using the constructor, as can be seen from the declaration of the constructor shown in

Section 5; the last five parameters of the constructor are the defects described here (with the same name used here in the declaration of the functions); the default situation has no defects. But this procedure can be quite tedious, since all the parameters before the last five must be given explicitly. Maybe it is better, for simplicity of use and readability, to give only the first 9 mandatory parameters to the constructor, and modify afterwards all the other parameters which are different from the default.

The reading functions are also defined, which read the values in the object:

```
double opt_simulation::get_ErrorDefoc_x();
double opt_simulation::get_ErrorDefoc_y();
double opt_simulation::get_ErrorGaussTrasv();
double opt_simulation::get_ErrorGaussLong();
int opt_simulation::get_ErrorNumberHarm();
double opt_simulation::get_ErrorHarm(int indexHarm);
```

`indexHarm` is the index in the array `CoeffHarm` of the Fourier coefficient that one would like to know; it can go from `0` to `NumberHarm-1`.

12. Shadowing

It can happen that the collector is partially shadowed by obstacles or by the adjacent string of collectors. This situation does not require a modification of the object `opt_simulation`, but it can be handled directly at the moment of the simulation using the two last parameters of the function `simulate`:

```
void opt_simulation::simulate(double x, double y, double z,
   double shadow_sx = 0, double shadow_dx = 0);
```

The parameter `shadow_sx` is the *fraction* of the collector width shadowed towards the left edge (where *x* `= -MirrorWidth`), and `shadow_dx` is the fraction of the collector width covered towards the right edge (*x* `= MirrorWidth`). As an example, `shadow_sx=0.5` means that the part of collector with `–MirrorWidth<`*x*`<0` is shadowed, and the rays are shot only on the other half.

The “fraction of collector width” is referred to the opening of the collector (the plane *y*=`MirrorWidth`^{2}/*f*). So, if we set `shadow_dx=0.1`, the rays that are not shot due to the shadowing are the ones that cross this plane between *x*`=0.8*MirrorWidth` and *x*`=MirrorWidth`.

As an example, we can consider a field of collectors, perfectly collimated, with a spacing of 1.5 times their openings between rows of collectors. Suppose that the sun is low on the horizon and that the collectors must be tilted of 60° with respect to the vertical direction, to face the sun correctly. In this case, each collector will shadow exactly a quarter of the collector behind it, as shown in Figure 7. This situation can be reproduced by setting `shadow_dx=0.25`:

```
simul.simulate(0,cos(Incl),-sin(Incl), 0, 0.25);
```

(`Incl` is the inclination of the sun with respect to the collector plane). With this command, only the solar beams in the first three quarters of the collector will be shot.

If the two shadowing parameters are omitted, they are set to the default value 0, which means that the collector is completely exposed to the sun.

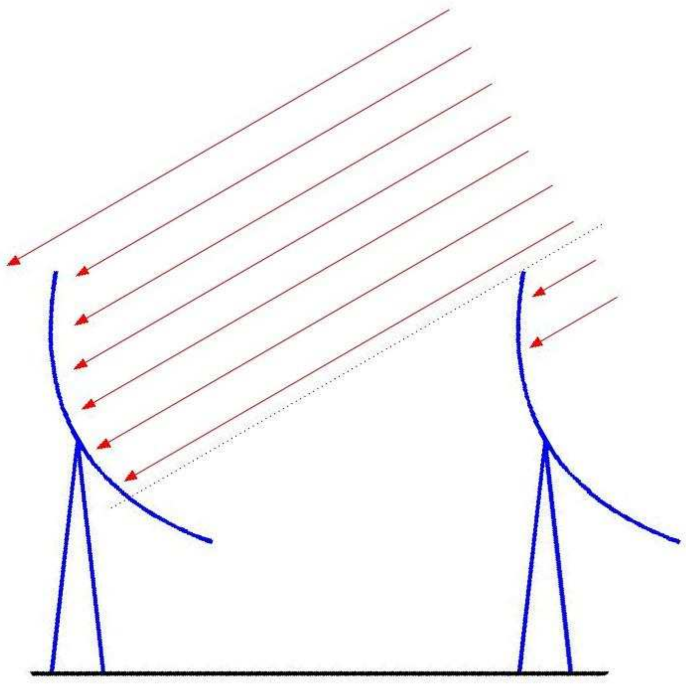


Figure 7: Shadowing of a collector by a parallel row of collectors.

13. Final Example

Now a final, long example will be shown, using most of the features previously discussed. Also, two additional optical functions will be defined at the beginning, and then they will replace two of the functions defined in the header `opt_fun_definition.h`. The detailed explanation of the program is given in the comments.

```
#include<iostream>
#include<fstream>
#include<cmath>

#include"SimulTrough.h"
#include"opt_fun_definition.h"


// a reflectivity function depending on the wavelength is defined
// (it will be used in the third simulation):
double reflect_new(double cosinc, double lambda) {
       return .05+.05*lambda/10000;
}

// a refractivity function depending on the wavelength is defined
// (it will be used in the third simulation):
double refract_new(double lambda) {
       return 1.5+.1*lambda/10000;
}


using namespace std;

int main() {

// initialization of the optical functions:

   OPT_FUN_INIT;


// definition of the geometrical parameters:

   int NumSurface = 3;

   double Focus = 1.8,
         MirrorWidth = 3.0,
         FreeSpace = 0.05;

   double RadiiReceiver[] = {0.065, 0.062, 0.035};

// the object simul is built:

   opt_simulation simul(Focus, MirrorWidth, FreeSpace, NumSurface,
         RadiiReceiver, ReflectMir, ReflectSurfaces, RefInd,
         Absorption);


         // ***** first simulation *****

   cout<<"---- FIRST SIMULATION ----"<<endl<<endl;

// tracking error of 0.01 rad:

   double ErrorColl = .01;
   simul.simulate( -sin(ErrorColl), cos(ErrorColl), 0 );


// the energy distributions are saved in files:

   ofstream of("distrib_absorber_1.dat");
   ofstream of2("distrib_glass_1.dat");

   for (int i=0; i<72; i++) {
         of<<simul.get_absorber(i)<<endl;
         of2<<simul.get_absorb_glass(i)<<endl;
   }

// output the total efficiency

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;

// the control error is printed:

   cout<<endl<<"Control error = "<<
         simul.get_rec_error()<<endl<<endl<<endl;
         // ***** second simulation *****

   cout<<"---- SECOND SIMULATION ----"<<endl<<endl;

// printing of the simulation progress is disabled

   simul.printoff();

// the width of the mirror is reduced to 4 m, and the absorber is
// set to 8 cm of diameter instead of 7:

   simul.set_MirrorWidth(2.0);
   simul.set_RadiiReceiver(2,0.04);

// the precision is enhanced: the sun is divided in 20 annuli
// and 500 beams are shot

   simul.set_nAnnuli(20);
   simul.set_nStep(500);

// after these changes, we check the number of rays of a solar
// beam printing it on the screen:

   cout<<"Rays of a solar beam = "<<
         simul.get_nSolarRay()<<endl<<endl;

// since the number of rays has been increased, we can change the
// resolution of the energy distribution figure on the receiver:
// we double the number of sectors (144 sectors of 2.5°)

   simul.set_nSector(144);

// the receiver is out of focus: 2 cm below and 3 cm at the right
// of the correct focus

   simul.set_ErrorDefoc(0.03, -0.02);

// the mirror surface is also slightly irregular, with a standard
// deviation of 0.005 rad with respect to the correct inclination
// in the transverse section:

   simul.set_ErrorGaussTrasv(0.005);

// inclination of the sun: 30°, no tracking error

   double Incl = 3.141592/6; // =PI/6

// we also suppose that the final quarter of the collector opening
// is shadowed by some obstacles:

   double sh_dx=0.25;

// simulation:

   simul.simulate( 0, cos(Incl), -sin(Incl), 0, sh_dx );

// the distributions of absorbed energy are printed, using
// the function get_nSector() to know how many sectors should be
```

```
// listed and how wide they are

   cout<<"Energy distribution on the absorber:"<<endl;

   int num_Sector=simul.get_nSector();
   double width_Sector=360/num_Sector; // width of an angular sector
                                       // in degrees

   for (int i=0; i<num_Sector; i++) cout<<
         "Energy absorbed between "<<width_Sector*i<<
         "° and "<<width_Sector*(i+1)<<"° = "<<
         simul.get_absorber(i)<<endl;


// the distribution of the energy absorbed by the glass is
// also listed:

   cout<<endl<<
         "Distribution of the energy absorbed by the glass:"<<endl;

   for (int i=0; i< num_Sector; i++) cout<<
         "Energy absorbed between "<<width_Sector*i<<
         "° and "<<width_Sector*(i+1)<<"° = "<<
         simul.get_absorb_glass(i)<<endl;

// the total efficiency is printed

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;

// the control error is printed:

   cout<<endl<<"Control error = "<<
         simul.get_rec_error()<<endl<<endl<<endl;

// the distributions of energy are saved in files:

   ofstream of3("distrib_absorber_2.dat");
   ofstream of4("distrib_glass_2.dat");

   for (int i=0; i<num_Sector; i++) {
         of3<<simul.get_absorber(i)<<endl;
         of4<<simul.get_absorb_glass(i)<<endl;
   }


// ***** third simulation *****

   cout<<"---- THIRD SIMULATION ----"<<endl<<endl;

// we use the same geometry as in the second simulation,
// but we remove the defects previously introduced:

   simul.set_ErrorGaussTrasv(0);
   simul.set_ErrorDefoc(0,0);

// we introduce instead a double-focus deformation:

   double harm[] = {0,0,0,-0.01};
   simul.set_ErrorTrasv(4,harm);
```

```
// we change the reflectivity function of the external face of
// the glass (element 0 of the array) with the new function
// defined:
// we first associate the function with a pointer of the correct
// type (reflect_s) and then pass it to the function
// set_ReflectSurface:

   reflect_s newrefl = &reflect_new;
   simul.set_ReflectSurfaces(0,newrefl);

// we also change the refractive index of the glass, introducing
// wavelength dependence (the glass is the medium 1: the order is
// air, glass, vacuum)

   refrac_index_function newrefrac = &refract_new;
   simul.set_RefracIndex(1, newrefrac);

// since we introduced wavelength dependence in some optical
// functions, it is advisable to enable wavelength analysis: we
// choose to shoot 10 rays of different wavelengths for each solar
// ray of the beam

   simul.set_nLambda(10);

// to save time, we reduce the precision

   simul.set_nAnnuli(10);
   simul.set_nStep(200);
   simul.set_nSector(72);

// we enable the process printing:

   simul.printon();

// simulation, with normal incidence and no shadowing:

   simul.simulate(0,1,0);

// the total efficiency is printed

   cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;

// the control error is printed:

   cout<<endl<<"Control error = "<<
         simul.get_rec_error()<<endl<<endl<<endl;

// the distributions of energy are saved in files:

   num_Sector = simul.get_nSector();

   ofstream of5("distrib_absorber_3.dat");
   ofstream of6("distrib_glass_3.dat");

   for (int i=0; i<num_Sector; i++) {
         of5<<simul.get_absorber(i)<<endl;
         of6<<simul.get_absorb_glass(i)<<endl;
   }
```

```
// ***** fourth simulation *****

   cout<<"---- FOURTH SIMULATION ----"<<endl<<endl;

// here, a more drastic change is made: the receiver is uncovered,
// and the number of concentric surfaces changes from 3 to 1. New
// pointers for the optical functions must be consequently defined.
// For the absorber, we use the same reflectivity function defined
// in opt_fun_definition.h, and called ReflectAbsorber (see
// Section 5 of the user guide). The arrays of the indices
// of refraction and of the absorption functions are now only
// composed of one element (the air); we use the same functions
// defined in opt_fun_definition.h (RefIndex_air
// and Absorption_air).

   double newRadiiReceiver[] = {0.04}; // new array of radii

// new arrays of optical functions:

   reflect_s NewArrayRefl[1];
   NewArrayRefl[0] = &ReflectAbsorber;
   refrac_index_function NewArrayRefrac[1];
   NewArrayRefrac[0]= &RefIndex_air;
   absorption_function NewArrayAbsorption[1];
   NewArrayAbsorption[0]= &Absorption_air;

   simul.set_NumSurface(1, newRadiiReceiver, NewArrayRefl,
         NewArrayRefrac, NewArrayAbsorption);

// since none of the optical functions has wavelength dependence,
// the wavelength analysis can be disabled:

   simul.set_nLambda(1);

// we eliminate the double-focus error
   simul.set_ErrorTrasv(0,0);

// now we have a simple system with naked receiver and without
// defects; since the system is simple (fast calculation)
// we enhance the precision:

   simul.set_nAnnuli(30);
   simul.set_nStep(1000);

// sectors of only 1°

   simul.set_nSector(360);

// simulation, with normal incidence and no shadowing:

   simul.simulate(0,1,0);

// the distribution of energy absorbed by the absorber surface is
// listed, using the function get_nSector() to know how
// many sectors should be listed and how wide they are

   cout<<"Energy distribution on the absorber:"<<endl;
```

```
      num_Sector=simul.get_nSector();
      width_Sector=360/num_Sector;  // width of an angular sector
                                    // in degrees

      for (int i=0; i<num_Sector; i++) cout<<
            "Energy absorbed between "<<width_Sector*i<<
            "° and "<<width_Sector*(i+1)<<"° = "<<
            simul.get_absorber(i)<<endl;


   // the distribution of the energy "absorbed by the glass" is also
   // listed: in this case there is no glass, and the library manages
   // this situation returning a list of zeros

      cout<<endl<<
            "Distribution of the energy absorbed by the (absent)
            glass:"<<endl;

      for (int i=0; i< num_Sector; i++) cout<<
            "Energy absorbed between "<<width_Sector*i<<
            "° and "<<width_Sector*(i+1)<<"° = "<<
            simul.get_absorb_glass(i)<<endl;

   // the total efficiency is printed

      cout<<endl<<"Efficiency = "<<simul.get_efficiency()<<endl;

   // the control error is printed:

      cout<<endl<<"Control error = "<<
            simul.get_rec_error()<<endl<<endl<<endl;

   // the energy distribution is saved in a file:

      ofstream of7("distrib_absorber_4.dat");

      for (int i=0; i<num_Sector; i++) {
            of7<<simul.get_absorber(i)<<endl;
      }

   }
```

At the end of the execution, there will be 7 files containing the data on the energy distributions, four (`distrib_absorber_1.dat`, `distrib_absorber_2.dat`, `distrib_absorber_3.dat`, `distrib_absorber_4.dat`) containing the distribution of the energy usefully absorbed by the receiver, and three (`distrib_glass_1.dat`, `distrib_glass_2.dat`, `distrib_glass_3.dat`) with the energy absorbed by the glass. The data can be plotted with graphical applications and compared, to see the effect of the parameter changes, of the shadowing and of the position of the sun.

This table is a summary of the four simulations of the example, with the time required on a Pentium 4, 3.06 GHz machine, and the resulting efficiency:

	Geometric properties	Defects	Simulation parameters	Sun position	Simulation time	Efficiency
Simulation 1	Coll. Width: 3 m Surfaces: 3 Radii: 6.5 cm, 6.2 cm, 3.5 cm	None	Steps: 200 Annuli: 10	(-sin 0.01, cos 0.01, 0) (tracking error)	10 s	57.62%

Simulation 2	Coll. Width: 2 m
	Surfaces: 3
	Radii: 6.5 cm, 6.2 cm, 4 cm
		Transverse random error;
		defocalisation;
		shadowing
			Steps: 500
			Annuli: 20
				(0, cos π/6, sin π/6) (sun inclination)	135 s	23.33%

Simulation 3	Coll. Width: 2 m Surfaces: 3 Radii: 6.5 cm, 6.2 cm, 4 cm	Double-focus deformation	Steps: 200 Annuli: 10 Wavelength analysis (10 wavelengths)	(0, 1, 0)	105 s	47.25%
Simulation 4	Coll. Width: 2 m Surfaces: 1 Radius: 4 cm	None	Steps: 1000 Annuli: 30	(0, 1, 0)	65 s	84.58%

The following figures (Figure 8-11) show the distributions of energy obtained.

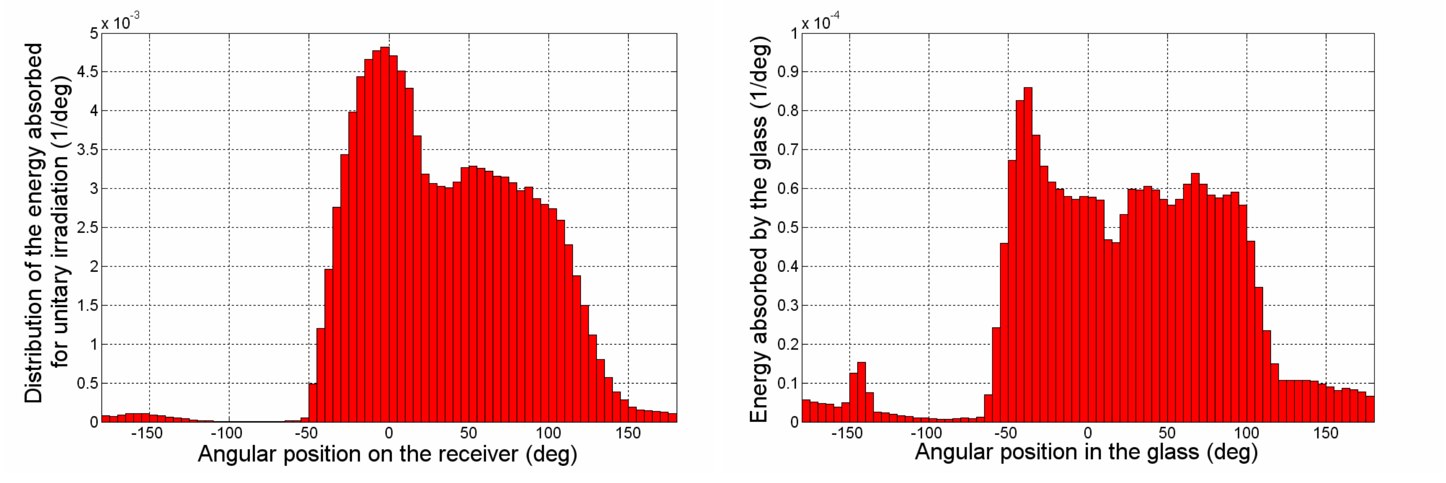


Figure 8: Distributions obtained from the first simulation.

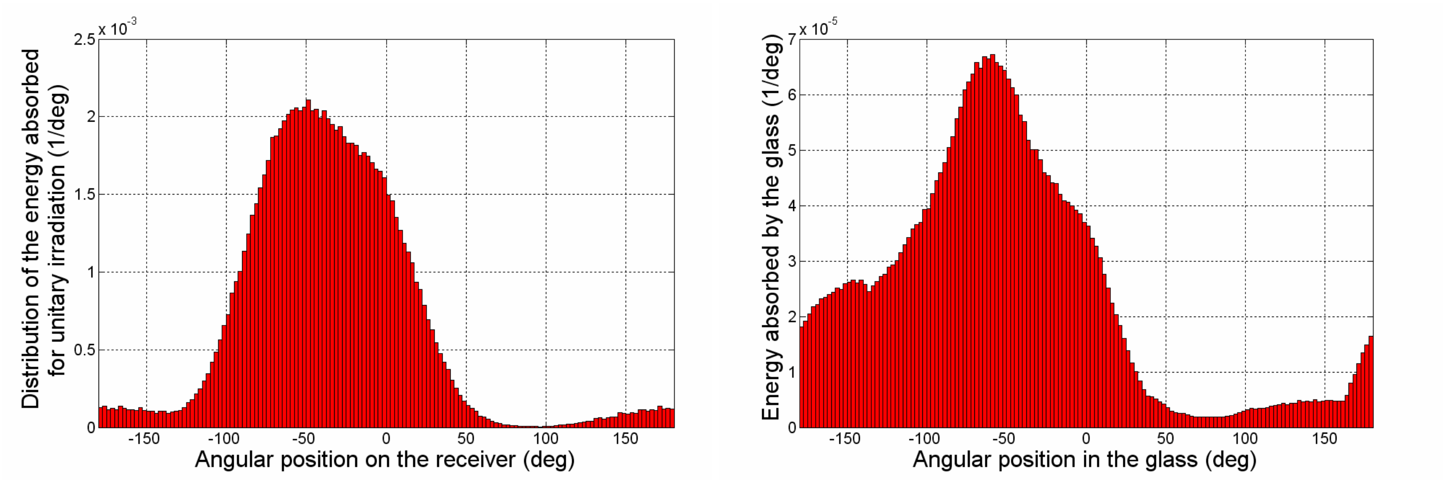


Figure 9: Distributions obtained from the second simulation.

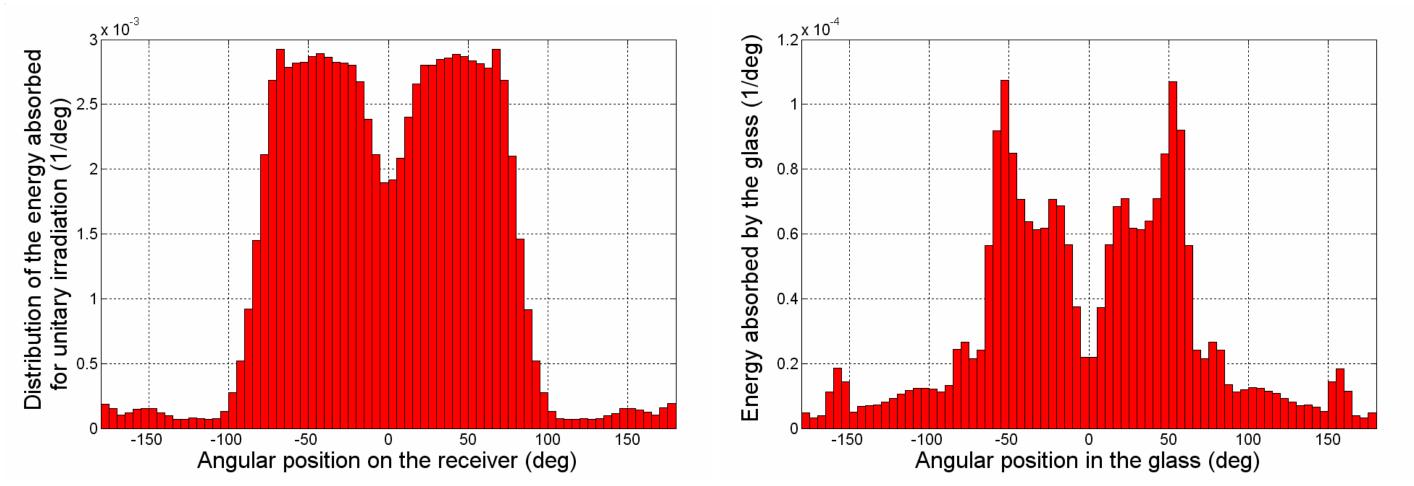


Figure 10: Distributions obtained from the third simulation.

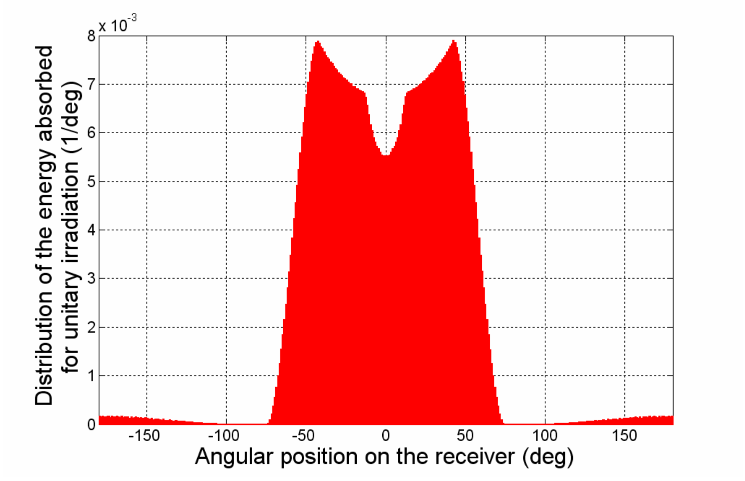


Figure 11: Distribution obtained from the fourth simulation.

Appendix: GNU General Public License

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <http://fsf.org/>
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

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.