COPHASAL™

Optical Design Software

User's Manual

XLOPTIX LLC
www.xloptix.com
cophasal_support@xloptix.com

Version 1.7.9

July 28, 2025

Notice

Trademarks

Disclaimer

Contents

Release Note

Welcome to the introductory version of the COPHASAL optical design software. COPHASAL is a high performance polarization ray tracing core that is wrapped in a command line interface based on the Lua programming language[8] and which is further enhanced by other functionality to support optical design.

In its currant avatar, there is only the command line interface. A graphical user interface is planned down the road. And although the inbuilt analysis functions are limited, they are sufficient for designing many optical systems. Further more, the scripting interface enables the user to extend the functionality. Some examples are located in the examples sub folder. Lua scripts are readily recognized by various code editors for syntax highlighting and AI coding assistants can greatly enhance the experience.

For this version:

- Display of the optical system and plots is done in the browser.
- Command line and scripts based interface.

Many development tasks related to user interface, analysis functions, etc. are in the pipeline. Often used functions will be incorporated in the high performance core with many implemented in user space scripts. However, the development going forward will be progressively dictated more and more by your requirements. Please write to us and let us know your suggestions, requirements, and any issues that you encounter. You can always reach us at cophasal_support@xloptix.com.

All changes since version 0.0.0 are listed on the next pages.

Additions

1. CONIC, conic surface
2. QCON, QCON polynomial based surface
3. TORIC, toroidal surface
4. `change_surf()`, this function changes the surface type, maintaining common parameter states
5. `check_globals()`, function to (un)set strict global variables checking
6. Elegant CLI, compact command line interface has been introduced
7. `FOCAL_MODE`, new parameter in system settings to support focal modes like `"AFOCAL"`

Updates

1. Undo/Redo, stop surface was shifting upon Undo/Redo actions
2. `os.exit()`, now exits gracefully
3. `first_order()`, large gamma rotation between start and stop surface results in systematic error in ABCD matrix estimation
4. `REPOS`, reverse repositioning mode now places the surface and then applies the repositioning
5. `freeze()`, supports freezing whole system
6. `del_tol()`, supports deleting all tolerances
7. `profile_data()`, updated to include derivative information
8. `showlens()`, 3D display supports non-rotational surfaces, input table keys are checked
9. Updated how reference rays are launched to allow curved object surface (`"S 1"`)

Chapter 1

Introduction

COPHASAL is a ray tracing based optical modeling and analysis software that can also help with design and optimization.

Unless otherwise noted, a boldfaced lower case character represents a vector ($\mathbf{v}$) while a boldface upper case character represents a matrix ($\mathbf{M}$), and an individual element will be represented by the corresponding non-boldface character with element index in the subscript and within parenthesis. For matrices, the first index is the row number ($M_{(r,c)}$). A complex quantity will be represented by tilde on top ($\tilde{c}$).

1.1 Conventions

The conventions used by COPHASAL are based on the book Principles of Optics[1]. To summarize, the evolution of the electric field of harmonic plane wave is described by the following relationship

$$\tilde{\mathbf{e}}(\mathbf{r}, t) = \mathbf{e_0} \times e^{i(\tilde{\mathbf{k}}.\mathbf{r} - \omega t - \phi)} \tag{1.1}$$

The complex index of refraction is $\tilde{n} = n + i\kappa$, where $\kappa > 0$ for absorbing medium.

1.1.1 Polarization

Consider a ray traveling along the z direction, $\tilde{e}_{(z)} = 0$. If $\tilde{e}_{(y)}/\tilde{e}_{(x)} = -i$ then we have a right circular polarization.

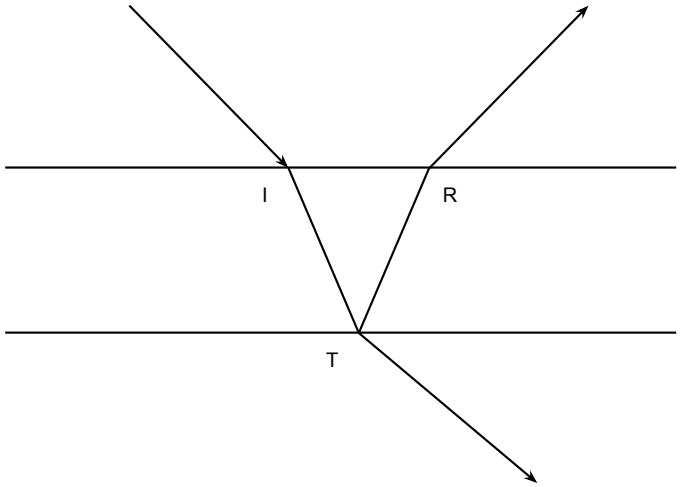

Figure 1.1: Ray is incident on an interface with a thin film stack containing a single layer. The incident ray end at the point marked by I, reflected ray starts from R and transmitted ray starts from T.

1.1.2 Thin Film

When tracing rays across thin film stacks, COPHASAL breaks the ray path. This is described in the following figure. The geometric phase of the rays is based on the points I, R and T.

The thin film phase as calculated by the characteristic equation of the film stack is added as follows. The phase term common to the s and p polarization is removed and accounted for by augmenting the geometrical phase term in the ray. The remaining phase term is used to create the Jones matrix in the s-p coordinate system and applied to the electric field.

1.2 Units

Units of various quantities in COPHASAL is listed in table 1.1

Quantity	Units
Dimensions	Millimeter
Construction angles	Degree
Wavelength	Micrometer
Layer thickness	Nanometer
Temperature	Centigrade
Pressure	Atmosphere
Relative Humidity	%

Table 1.1: Units for various quantities.

1.3 System Requirements

The core of COPHASAL can be adapted to a particular architecture if needed. However, it has been tested on the following 64 bit architectures.

OS	Processor
Windows 11	Intel processor (x86)
OSX 13 or later	Apple Si (ARM)

Apart from this, please note that the default web browser must be set to a browser based on the Chromium project, like Chrome. This is because COPHASAL generates web pages on the fly and attempts to open them in the browser. This is needed for display of 3D graphics and plots.

1.4 Installation

Download and run the installation file COPHASAL_######.exe to install COPHASAL. The version string is included in the installer file name. The installation folder can also be added to the system paths environment variable so that the executable cophasal.exe can be opened in a terminal application, from any folder.

1.4.1 License

The license file is called *license* and it is placed in the folder with the COPHASAL executable. The licensing scheme is simple, there are two licenses. The default license is for non-commercial use, for example by stu-

dents. For commercial use, please obtain the commercial license from XLOPTIX LLC.

The two licenses differ only in the welcome banner that is displayed at the start of the application. The banner identifies the type of license in use. Its just a reminder that in case of commercial use, the appropriate license must be maintained.[1]

1.5 User Interface

Command line interface of COPHASAL is essentially based on the Lua scripting language that has been augmented with additional functionality[2]. The prompt for the command line interface is `CPH>`. The default file extension is .lua and this makes these scripts ready for syntax highlighting by many code editors and also take advantage of available AI coding assistants. Unless otherwise noted, it will be assumed that the file extension is .lua.

1.6 User Setup

Within the application folder is provided a script called `"user_setup"`. This script is run every time the COPHASAL starts. It is strongly recommended that at the minimum this file be edited to include a command to change the application working folder to a folder in the user's home folder. An example of this file is shown in listing 1.1.

```
print "This is user setup file. "
print ""

-- Load the glass catalogs
add_glass_catalog"schott"

-- Listing precisions
set_numeric_precision(4)

-- cd, add_path, etc.
-- Change working folder.
```

[1] Installation and licensing mechanism is subject to change.

[2] Currently it is the only interface to COPHASAL. A graphical user interface will be added in subsequent releases.

```
cd "C:/Users/Your_Name/Documents/Project_Folder"
```

Listing 1.1: Reference listing of the user setup script.

Notice the last line in the listing, there is a function called `cd("<path>")` which takes a single string argument. In Lua, when a function takes a single string argument, the parenthesis are optional.

1.7 Support

COPHASAL is a fairly complex piece of code and even though it has gone through extensive testing, bugs are expected. Additionally, there are various features that are lined up to be added and additional improvement requests are generated on the go. If you encounter a situation that requires our attention, please contact us at cophasal_support@xloptix.com. Our engineers will reply back with a path forward or if needed, a live screen share session can be arranged.

Chapter 2

Quick Start Example

This example illustrates the basic process of realizing a simple optical system in COPHASAL. In this example, a singlet imaging lens will be designed. The commands that will be used in this example can be individually supplied to COPHASAL or assembled in a script file. The latter approach is the preferred method. We will use both methods in the following. Create a script file called "example".

2.1 Blank Lens

When COPHASAL starts it has a "blank lens" in the system. It is the starting point and in this state there are only three surfaces defined in the sequential surfaces list. The same state can be reached by issuing the command `blank()`. The optical system layout is shown in 2.1. There are three sequential surfaces. Sequential surfaces are identified by the letter S. S 1 is object, 1 mm from it and towards positive $\hat{z}$ direction is S 2 which is also the system stop. One more mm from the stop is the image surface.

Every surface has a surface local coordinate system and by default its $\hat{z}$ axis points towards right with the $\hat{y}$ axis pointing up. For the sequential surfaces, surface thickness usually positions the next surface along the local $\hat{z}$ axis at a displacement equal to the thickness, unless the next surface is repositioned or its coordinate system globally refers to another surface.

In the layout graphics, the $\hat{x}$ and $\hat{y}$ axis of every surface coordinate system is marked by a small red and green line that meets at the origin of that coordinate system. In figure 2.1, the first vertical green line is object $\hat{y}$ ($\hat{x}$

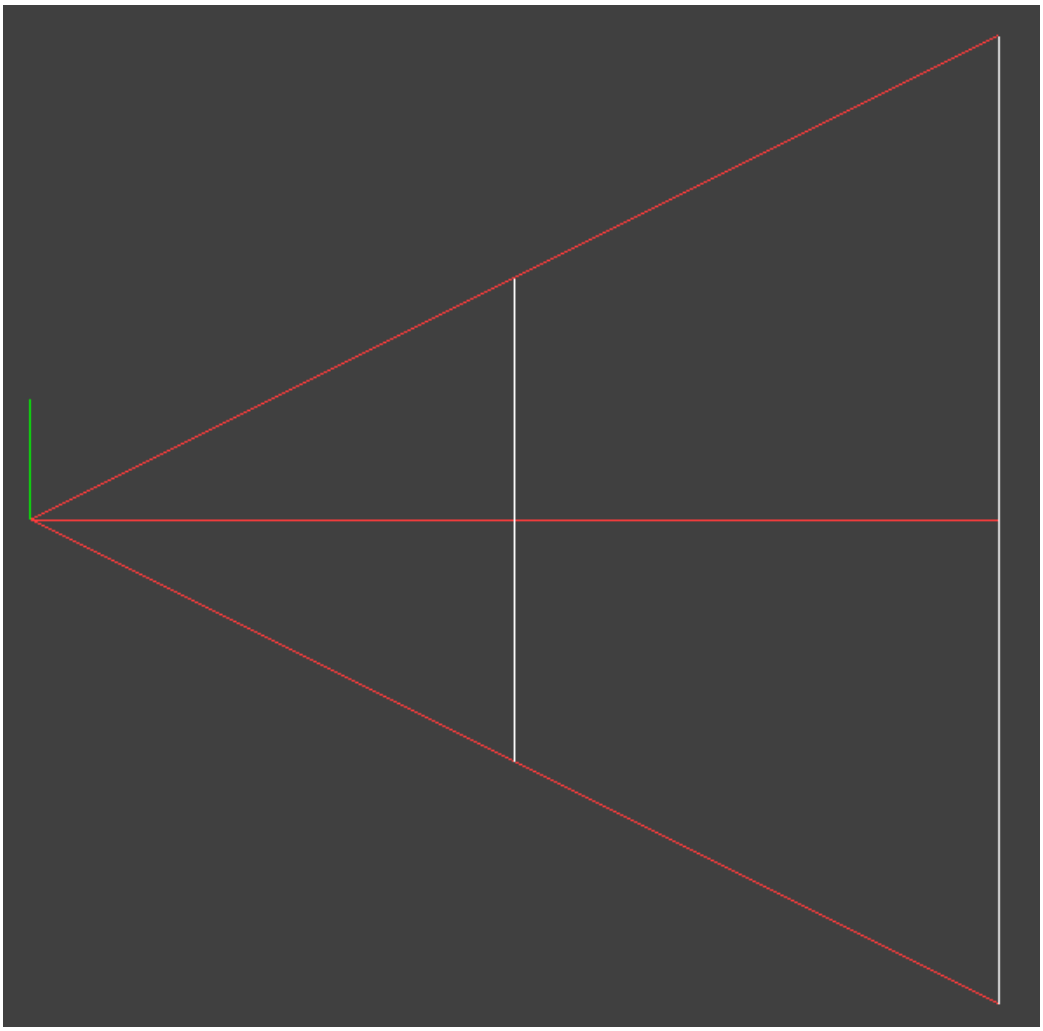

Figure 2.1: Optical system after `blank()`. Use the command `showlens()` to display the layout.

is pointing into the screen). The white vertical line for S 2 and image is indicating the Y-Z cross-section of the surface and the active aperture on that surface.

After `blank()`, field is in terms of object height and there is only one on axis field defined. The pupil is specified as entrance pupil diameter (EPD) of 1 mm. The red rays in figure 2.1 are the chief (parallel to $\hat{z}$) and marginal rays for this on axis field. Similarly, `blank()` results in only one wavelength defined for the system and it is set to be equal to 0.5 microns.

2.2 Singlet Start

So the first entry in our example script is going to be `blank()` just so that we have a known starting point. Here are the contents of the script to realize a starting singlet lens.

```
blank()

-- Wavelengths.
set("wl w 1", 0.6) -- Set the wavelength.
-- w[1].wl = 0.6; does the same thing as above.
ins_wl(2, 0.5) -- Insert wavelength number 2
ins_wl(3, 0.4)

set_pwl(2) -- Designate primary wavelength

-- System.
set("p_val", 10) -- Pupil type is already EPD
                 -- Set its value to 10 mm
-- Feilds.
ins_fld(2) -- Field type is angle
set("y f 2", 7) -- 7 degrees
ins_fld(3) -- Now we have three fields
set("y f 3", 15)

-- Surfaces.
set("thi s 1", 1e11) -- Object is far away
set_label("s 1", "o")

ins_surf("s 2")
```

```
-- Label the surface, lens first surface
set_label("s 2", "l1")
-- Set radius of curvature, notice the use of label
set("rdy s 'l1'", 50) -- same as: s['l1'].rdy = 50
set("thi s 'l1'", 5)
set("gls2 s 'l1'", "n-bk7|schott")
-- Note how catalog glass is applied. "|" is used to
-- separate the glass name and the catalog name. Every
-- surface defines a boundary between two mediums,
-- GLS1 and GLS2. For sequential surfaces GLS1 is
-- automaticlly set. One only needs to set GLS2 and
-- think of it as the glass following the surface.
-- Default glass material is the medium set, which is
-- "AIR" by default.

ins_surf("s 3")
set_label("s 3", "l2")
set("rdy s 'l2'", 200)
set("thi s 'l2'", 15)

set_label("s 4", "s") -- Stop surface
set("thi s 's'", 100)

set_label("s 5", "i")
```

Listing 2.1: Example script to realize a singlet lens.

To run this script, issue the command `runfile("example")`. Use `showlens()` to display the lens layout in the browser. The layout is shown in 2.2. Surface details can be listed by using the command `list("S 2")`. However to list the important information of all the defined surfaces, omit the surface identifier, `list"s"`.

```
S:
 Z1:     LBL       TYP       RDY          THI          MED          RFR      AUT_APE...
 S 1       o    SPHERE         0        1e+11          AIR     TRANSMIT             0
 S 2      l1    SPHERE        50            5    N-BK7|SCH     TRANSMIT         10.64
 S 3      l2    SPHERE       200           15          AIR     TRANSMIT         9.615
 S 4       s    SPHERE         0          100          AIR     TRANSMIT         4.248
 S 5       i    SPHERE         0            0          AIR     TRANSMIT         32.65
Stop: 4
```

Listing 2.2: Listing of all the sequential surfaces so far.

Let us look at how the various parameters of the system are addressed

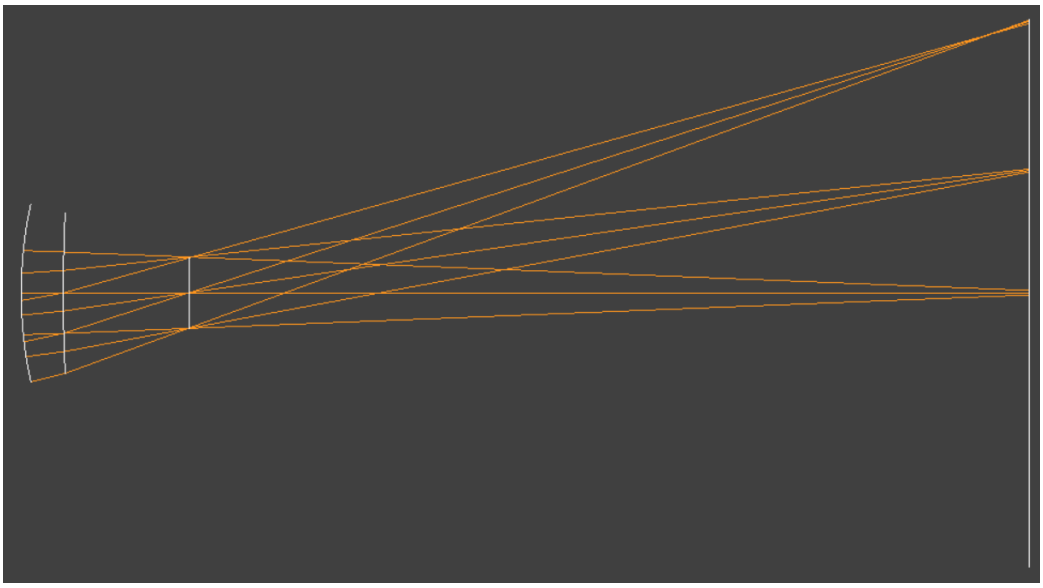

Figure 2.2: Layout of the singlet lens at the starting.

in the listing 2.1. Every parameter has a name and is associated with a numbered item in a list. For example `"thi s 2"` is the thickness parameter of the second sequential surface. In fact, the same scheme if also applicable for thin film stacks. The only difference is that the film name is also part of the parameter identification, for example `"THI MgF2 L 1"`.

2.3 Singlet Start Performance

The `spot_data()` function is used to estimate the root mean square spot size for the system. The plotting function is `plot()` and it can be used for generating the spot pattern.

```
-- Analysis, get spot data
local data = spot_data() -- Default is F 1, W = PWL
plot(data.spots, {type = "scatter",
                  labels = {"PWL, F 1"}
                 }
    )
-- Note the {} above, in Lua this is a table
-- containing key-value pairs etc. More on this later.
```

```
print("RMS spot size: ", data.rms)
```

Listing 2.3: Listing of all the sequential surfaces so far.

Listing 2.3 shows how these functions are used to generate the spot pattern and print the rms spot size. Note the use of the Lua keyword `local`, this declares the variable to be local. Without this qualification, the default in Lua is to declare variables in the global scope. Please make it a habit to use `local` to declare variables otherwise the global name space will get unnecessarily crowded. The generated spot diagram is shown in figure 2.3

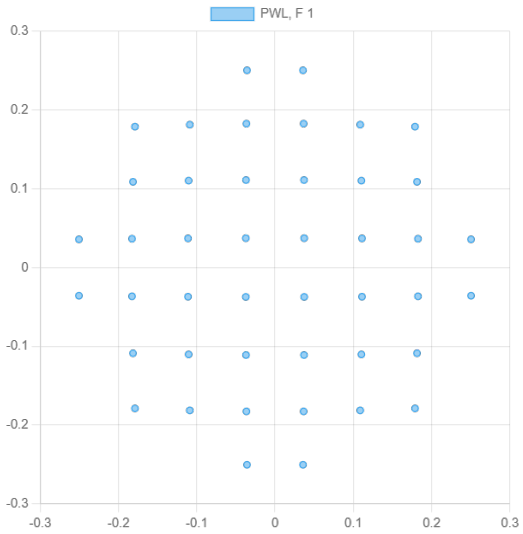

Figure 2.3: On axis spot diagram of starting system.

and `RMS spot size: 0.19195509414023` is printed on the screen.

2.4 Optimization

```
-- Define variables.
vary("cuy s 'l1'")
vary("cuy s 'l2'")
vary("thi s 'l2'")
vary("thi s 's'")
vary("cuy s 'i'")
```

```
-- Internally, its only curvatures. RDY is only for
-- convenience for set and get functions.

-- Our optimizers are called Solvers and they can be
-- declared as local or global.
local o = Solver.new()
o:set_cost(get_cost)
-- get_cost() is the inbuilt cost function
o:set_rel_tol(1e-2) -- Set relative tolerance

o:solve() -- Optimize
```

Listing 2.4: Define variables and solver.

For optimization, we must have some parameters of the system designated as variables. This is done by the function `vary()`. The optimizer is actually a user type called `Solver` and it must be first declared, setup, and then its `solve()` function is called for optimization. In listing 2.4, notice how the local solver's internal functions are called using the ":" operator.

2.5 Final Performance

After optimization, the RMS spot size printed by the script is as follows: `Final RMS spot size:  0.0089595633177525`. For the optimized system, layout is shown in figure 2.4, spot diagram is in figure 2.5, and the surface listing is in 2.5

```
S:
 Z1:     LBL       TYP       RDY          THI          MED          RFR      AUT_APE...
 S 1       o    SPHERE         0        1e+11          AIR     TRANSMIT            0
 S 2      l1    SPHERE     49.36  v         5    N-BK7|SCH     TRANSMIT        11.03
 S 3      l2    SPHERE     216.5  v     15.94  v       AIR     TRANSMIT        10.01
 S 4       s    SPHERE         0        100.6  v       AIR     TRANSMIT        4.183
 S 5       i    SPHERE    -60.34  v         0          AIR     TRANSMIT        30.61
Stop: 4
```

Listing 2.5: Listing of sequential surfaces post optimization.

2.6 List Ray

Concise information about a ray can be listed using the function `list_ray`. By default it lists the first field chief ray for the primary wavelength (`{r`

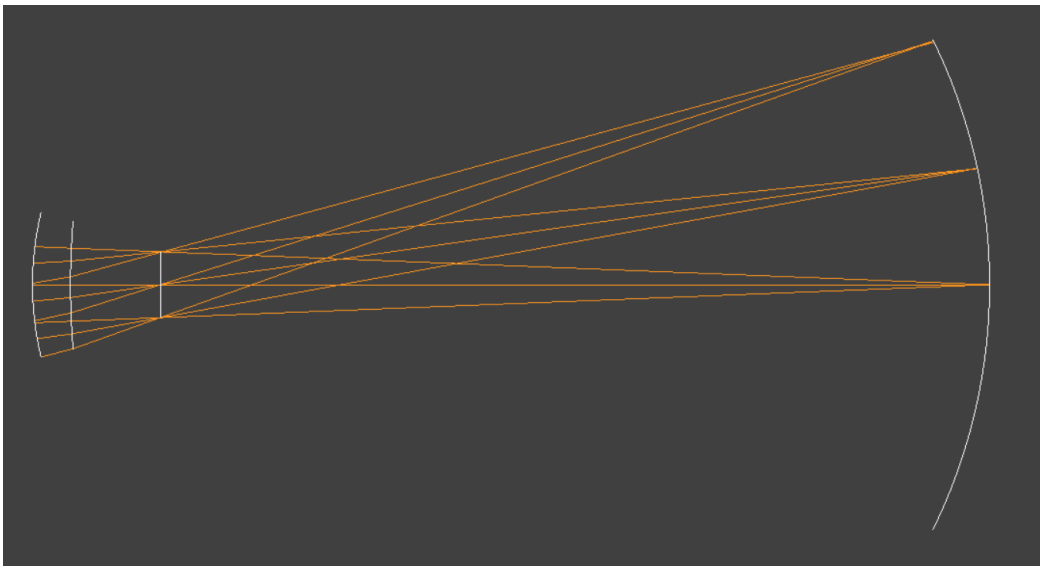

Figure 2.4: System layout after optimization.

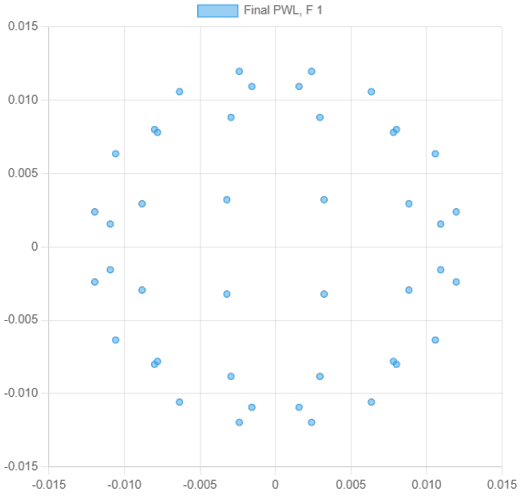

Figure 2.5: On axis spot diagram of optimized system.

= "ch", f = 1}). To list the +Y marginal ray information instead, call the function with the following argument `list_ray({r = "yp"})`. Listings 2.6 and 2.7 show the ray information for the chief and the marginal ray respectively.

```
SUR  X          Y          Z          L          M          GPH        T,R        A
S 2  0          0          0          0          0          0          0.9572     -
S 3  0          0          0          0          0          7.609      0.9572     0.9976
S 4  0          0          0          0          0          23.55      1          1
S 5  0          0          0          0          0          124.2      -          1
Intensity transfer: 0.9141
Jones vector: Yamp/Xamp: 0, Yphi-Xphi: 0
```

Listing 2.6: Ray listing for F 1 chief ray, PWL.

```
SUR  X          Y          Z          L          M          GPH        T,R        A
S 2  0          5          0.2539     0          0          0.254      0.9567     -
S 3  0          4.833      0.05394    0          -0.03483   7.563      0.9572     0.9977
S 4  0          4.175      0          0          -0.04136   23.46      1          1
S 5  0          0.01009    -8.437e-07 0          -0.04136   124.2      -          1
Intensity transfer: 0.9136
Jones vector: Yamp/Xamp: 0, Yphi-Xphi: 0
```

Listing 2.7: Ray listing for F 1 marginal ray, PWL.

2.7 Paraxial Image Location

In order to locate the image surface at the ideal location based on Gaussian optics, we have to make the thickness of the prior surface "react" to changes in the ideal image distance calculations. This is done by the function `pim_reaction(true)`. This will freeze the thickness and surface listing will show a "r" instead of a "v" next to this thickness.

Chapter 3

Lua Primer

The command line interface of COPHASAL is based on the Lua scripting language[8]. Lua is relatively simple yet versatile programming language and the detailed treatment of which is beyond the scope of this document. Only the basics of Lua relevant to this subject mater will be listed here. Please see the Lua reference manual (https://www.lua.org/manual/) if more details are needed. An excellent resource for Lua is the book "Programming in Lua"[5] written by one of the authors of the language itself. COPHASAL uses Lua version 5.4.6[1].

We can issue individual commands at the COPHASAL prompt or we can run a script file containing commands. In Lua parlance, these pieces of code, weather individual commands or the script file, are called "chunks". Let us supply COPHASAL with a simple chunk of code in a file called `test.lua`, with the contents listed in 3.1.

```
-- A double dash starts a comment
--[[A double square bracket starts a multi-line comment
    is not required here. ]]

local a = "a" -- Lua is case sensitive
-- a is local to the chunk, that is this script
```

[1]To facilitate Optics analysis and user experience, COPHASAL has introduced many additional functions. Apart from these, only a few changes have been introduced to base Lua and these are as follows. The user's global environment has been separated from Lua global environment. Lua function `dofile()` has been removed. Lua function `loadfile()` has been masked off and replaced by `runfile()` instead, it loads and runs the script.

```
B = "B" -- Global, list_gvr() to display all global items.

for i = 0, 2 do
    -- Lua is lexically scoped
    -- a is accessible here
    local A = "A"
    -- A is local only to the loop
    print(i .. " " .. A .. " " .. a)
end
```

Listing 3.1: Example script.

With this `test.lua` file saved in the working folder, issue the command `runfile "test"`, the following should be the result.

```
-- Result of runfile"test":
0 A a
1 A a
2 A a
-- While the command list_gvr() results in:
B                                       "B"
```

3.1 Global and Local Scope

An odd and important thing to note about Lua is that without limiting the scope of a variable to `local`, it is by default declared in global scope. Global variables are very useful and they can be used to pass information between two script runs. However, the global namespace should not be so populated that we run out of names. Hence, it is a good practice to always use `local` unless there is explicit need to declare the variable in the global scope.

3.2 Nil

In Lua, value of `nil` means nothing. When we assign `nil` to a variable, it will be marked for deletion and the Lua garbage collector will remove it from the variables list. In case some items are declared in the global scope that were supposed to be in local scope, then this approach can be used to delete them (example: `B = nil`).

3.3 Lexical Scoping

As can be seen from the listing 3.1, Lua follows lexical scoping. It means that for example in the case of nested function, the inner function will have full access to local variables of the enclosing function.

3.4 Types and Values

Lua is a dynamically types language. Variables can be assigned any value on the fly. There are eight basic types of variables in Lua. The function `type`

`Value`	`type(Value)`
true	boolean
3.141	number
"Hello"	string
`list_gvr`	function
`{}`	table
Solver	userdata

Table 3.1: 6 important Lua types and values

returns the string representing the variable type. The important types are listed in table 3.1. `userdata` will not be discussed in detail except for the fact that the `Solver`, the internal optimizer, is a `userdata`. Lua tables are very versatile and they are a heterogeneous collection of data, more later.

3.5 Basic Operations

The basic library functions are listed in table 3.2. They provide core functionality of Lua. Yet, many of them are encountered infrequently.

3.6 Logical Operations

There are three logical operators, `and`, `or`, and `not`. Note that both the value `false` and `nil` are treated as `false`, any thing else is treated as `true`. Hence the following results hold.

assert	collectgarbage	error
getmetatable	ipairs	load
next	pairs	pcall
print	rawequal	rawget
rawlen	rawset	require
select	setmetatable	tonumber
tostring	type	warn
	xpcall	

Table 3.2: Basic library functions.

```
nil or "a" --> "a"
nil and 10 --> nil

-- Very useful Lua idom:
x = x or v -- If x has a value, use it, else use v.
```

Both `and` and `or` use short-circuit evaluation that is the second operand will be evaluated only as necessary.

```
(condition1 and condition2)
-- If condition1 == false, condition2 not checked.
```

3.7 Arithmetic Operations

Operator	Name
+	addition
−	subtraction
*	multiplication
/	division
//	floor division
%	modulo (reminder)
^	exponentiation

Table 3.3: Arithmetic operators.

Table 3.3 lists the mathematical operators in Lua. Common mathematical functions are in the `math` library 3.4. It feels odd that `math.` has to be

attached to these functions names and although it is possible to not have to do this, it has been decided to keep Lua as original as possible.

math.abs	math.acos	math.asin
math.atan	math.ceil	math.cos
math.deg	math.exp	math.floor
math.fmod	math.huge	math.log
math.max	math.maxinteger	math.min
math.mininteger	math.modf	math.pi
math.rad	math.random	math.randomseed
math.sin	math.sqrt	math.tan
math.tointeger	math.type	math.ult

Table 3.4: Math library functions.

3.8 Relational Operations

Operator	Name
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal
==	equals
~=	not equals

Table 3.5: Relational operators.

Table 3.5 lists the relational operators in Lua.

3.9 Bitwise Operations

Table 3.6 lists the bitwise operators in Lua. These operators convert their operands to integers and operate on all bits. Shift operators fill vacant bits with zeros.

Operator	Name
&	AND
|	OR
~	exclusive OR
>>	right shift
<<	left shift

Table 3.6: Bitwise operators.

3.10 Strings

string.byte	string.char	string.dump
string.find	string.format	string.gmatch
string.gsub	string.len	string.lower
string.match	string.pack	string.packsize
string.rep	string.reverse	string.sub
string.unpack		string.upper

Table 3.7: String library functions.

Table 3.7 lists the functions from the `string` library. A string can be enclosed in double or single quotes. Quotes inside a string need not be escaped. However, for string inside another string, we will follow the convention that the primary string will be enclosed inside double quotes and the internal string in single quotes (example: `local str = "thi s 'stop'"`).

Just like multi line comments, matching double square brackets define a multi line string.

```
local code = [[
    local var = "hello world!"
    print(var)
]]
-- String variable "code" has two lines of string.
```

3.11 Tables

In Lua, tables are a versatile construct to realize a collection of data. Table data is always enclosed in braces. Here are some examples.

```
-- Declare a local table .
-- Table entries are key value pairs .
local tbl = {
    a = 1.0, -- key = a , value = 1.0
    b = {" x " , " y " , 3}, -- sequence
    c = {12 , 13 , 14} , -- array
    d = {} -- empty table
}

tbl.d[1] = "s 1"
tbl.d[#tbl.d + 1] = "s 2" -- tbl.d[2] = "s 2"
-- # is the table length operator.

print(tbl.a ) -- 1.0
print(tbl.b[2]) -- y
print(tbl.d[1]) -- s 1

list_tbl(tbl) -- list the table
```

We cannot just use the `print` function to list a table. Instead a function to list tables is provided, it is `list_tbl()`. Here is the table listing of the above defined table.

```
{
 d = {"s 1", "s 2"},
 c = {12, 13, 14},
 b = {" x ", " y ", 3},
 a = 1
}
```

Notice that the ordering of elements of `tbl` is not as expected but the ordering of elements of its sub tables is as expected. This is because we do not have control over how tables store data in the memory. However, if the keys are sequentially index from integers, as is the default case, then traversing them sequentially is trivial.

Tables with sequentially indexed integer keys are called lists. Starting index of 1 for lists is recommended. We say that there is a hole in a list if there is a `nil` value in the list that is not at the end. Lists without holes are called sequences and sequences with all values of the same type are called arrays.

Please note the difference between two kinds of table indexing. Consider

`tbl.a` and `tbl[a]`. The first one gets you 1.0 while the second one will try to index the table `tbl` at the value of the variable `a` which in the above example has not been declared.

Here are a few ways to traverse a table using `for` loop constructs.

```
local t = {1, list_tbl, x = 2, y = "A"}
for k, v in pairs(t) do
    print(k, v)
end
--> 1       1
--> 2       function: 0000017b573a24f0
--> x       2
--> y       A
-- Pairs function help traverse using key-value pairs.
-- k and v are loop local by default.

t = {1, list_tbl, 2, "A"} -- list
for k, v in ipairs(t) do
    print(k, v)
end
--> 1       1
--> 2       function: 0000017b573a24f0
--> 3       2
--> 4       A

-- More traditional for loop
-- Key goes from 1 to the length of table
for k = 1, #t do
    print(k, t[k])
end
-- Same output as above.
```

table.concat	table.insert	table.move
table.pack	table.remove	table.sort
table.unpack		

Table 3.8: Table library functions.

Table 3.8 lists the functions in the `table` library.

3.12 Functions

Functions must be defined before they can be used in the script. Here is a simple example of a script defining and using a function.

```
local function mysquare(x)
  return x * x + math.pi
end

local val = 10

print(mysquare(val)) --> 103.14159265359

-- Functions can return multiple values.
local n, k = indexo("s 1", 1) -- index of refraction
print(n, k) --> 1.0002722354781 0.0
```

Functions in Lua can be assigned new names on the fly and in fact they don't necessarily have to have any name. Lua also supports higher order functions, that is functions that take other functions as arguments and this is a very versatile feature. Consider the following example[5].

```
local function foo(x)
    return x + 1
end
-- Is equivalent to:
local foo2 = function(x)
    return x + 1
end

-- Table sort example
local tbl = {
    {name = "John", age = 23},
    {name = "Alice", age = 25},
    {name = "Bob", age = 22}
}

-- Higher order functions example
-- Sort by age
table.sort(tbl, function(a, b) return a.age < b.age end)

-- Print sorted table
```

```
for i, v in ipairs(tbl) do
    print(v.name, v.age)
end
--> Bob     22
--> John    23
--> Alice   25

local function compare_with(x)
    local temp = x

    return function(y)
        return temp == y
    end
end

local compare_with_5 = compare_with(5)
print(compare_with_5(5)) -- true
print(compare_with_5(6)) -- false
```

For simple data types (number, string, boolean), arguments are passed by value. This means that a copy of the value is passed to the function. Any changes made to the argument inside the function do not affect the original value outside the function.

Tables and full userdata (like the `Solver`) in Lua are always passed and assigned by reference. This means that when you pass these types as an argument, you're passing a reference to the original data, not a copy. Any changes made to the data inside the function will be reflected in the original data outside the function.

3.13 Control Structures

Lua offers the most common collection of control structures. We will only describe `if then else` conditional branching, `for` loops, `break` and `return` statements.

The following example illustrates the conditional branching in a function.

```
-- Function to test the sign.
local function signof(x)
  if x < 0 then
    return -1
```

```
  elseif x > 0 then
    return 1
  else
    return 0
  end
end
```

We have already looked at `for` loops. There are two kinds of `for` loops, numerical and iterator based. When we use the functions `pairs` and `ipairs` we are using the iterator based `for` loop. Examples of iterator based `for` loop are shown in section 3.11. Here is an examples of numerical for loop.

```
-- Numeric for loop syntax:
--[[
for i = start, stop, step do
    -- code
end

step is optional and defaults to 1
]]

for i = 1, 5, 2 do
    print(i)
end
-- Prints:
-- 1
-- 3
-- 5
```

The `break` statement is used to break out of a loop. We have seen the `return` statement, there is a subtlety regarding multiple returns. `return` can only appear as the last statement of a block. A block being a loop, for example. This is described in the following example.

```
-- Multiple return statement
local function foo(val)
    -- return here will be a syntax error
    if val > 0 then
        return "positive" -- OK
    elseif val == 0 then
        return "zero"     -- OK
    else
```

```
        return "negative" -- OK
    end
end
```

3.14 Understanding Error Messages

When COPHASAL encounters an error condition (arithmetic fault, inconsistent inputs, etc), it aborts the execution and prints a message indicating that a fault was detected. Consider the following listing of the contents of the file test.lua.

```
1 local args = {
2    sym = "none",
3    rays = {
4       f = "all",
5       w = "pwk"
6    }
7 }
8
9 showlens(args)
```

Upon running this file, we get the following output.

```
CPH> runfile"test"
runtime error: PWK: Unqualified item!
stack traceback:
        Path_To/scripts/display_helpers.lua:326: in function '
    showlens'
        Path_To/test.lua:9: in main chunk
        (...tail calls...)
        [string "runfile"test""]:1: in main chunk
```

The main error message is the first line that indicates that this is a runtime error. This is followed by a stack traceback listing. A stack traceback is sequential list of function calls that eventually led to this error. The first line is the final function call, that is the one encountering the error. The last line is the function that started it all, our command to run the test.lua file. According to this message, line 9 in test.lua is at fault. This is usually enough to pin point the issue, however in this case the actual mistake is on line 5, that is in the value of `args.rays.w`, it should have been `"pwl"`.

In conclusion, you've now successfully navigated the foundational concepts of Lua. With these building blocks, you're well-equipped to explore more advanced topics and begin crafting your own Lua scripts. The journey of mastering Lua is just beginning, and the possibilities are vast.

Chapter 4

Command Line Interface

The command line interface of COPHASAL is capable of customization. Some of the interface functions are written in Lua itself, like `list_grv()` and `editfile()` are defined in setup script located in the scripts folder[1]. Although Lua is case sensitive, we will adhere to the convention of using lower case keys for table entries. Following is a categorized list of all the additional functions defined in COPHASAL.

4.1 Parametric Actions

Parameters in COPHASAL have a name (like `thi`) and are usually associated with an item in a list. These items can be for example a surface in the sequential surfaces list. Item numbering starts with 1. These items and their string identifiers are listed in table 4.1. An example of a parameter name is `"cuy s 4"` (curvature of sequential surface number 4). Another example is `"rdy n 'edge'"` (radius of curvature of non-sequential surface that has been labeled as 'edge', no need to keep track of surface numbers). Here is an example for film stack glass parameter, `"GLS2 AR_COAT L 1"`, it is the glass name for layer one of a coating called "AR_COAT". Except for the label and coating names, these identifier strings and string parameter values are not case sensitive.

Some system level parameters are not associated with any item. These

[1]Global variables in Lua are defined in a table called `_G`. However, a simple "sandbox" has been implemented to hide this table from the user. Your global variables are actually in a table called `User_Globals`.

Item	String Identifier
Wavelengths	W
Fields	F
Sequential Surfaces	S
Non-Sequential Surfaces	N
This Film Layers	L
System Data	D

Table 4.1: Items and identifier strings.

are most of the System Data parameters like `PUPIL`, Wavelength parameters like `TEM` (temperature). Their identifier string is simply the parameter name.

Parameters in COPHASAL can be classified into three capabilities. These are the capability to *zoom*, *react*, and *vary*.

4.1.1 Zoomers

Many optical systems must be designed to simultaneously maintain specification under different arrangement of individual constituent components or parameter values. The most common example of such a system is the traditional Zoom lens that must maintain good imaging quality at all specified zoom positions. These zoom positions are realized by a specific arrangement or configuration of the component lenses making up the zoom lens. Such systems can be realized in COPHASAL by setting a different value for the constructional parameters of the system, for the different zoom positions. Zoomers are the parameters that support this capability. To zoom something is to have it represent potentially different values for different configurations (zoom positions). This is indicated by the letter "z" next to that parameter listing. The following are the system level functions to support the zoom capability.

But first, here is code block describing how to read the syntax language.

```
--[[
ret = func(arg1, [arg2], [arg3])

ret: return value
arg1: first argument
arg2: optional second argument, default: 0
arg3: optional third argument, default: {
```

```
  key1 = num,
  key2 = str,
  [key3] = 2,
  key4 = (option1 | option2)
}

Additional description: For arg3, key1 is required number, key2 is
    required string, key3 is optional number with default 2
and key4 is required with value either option1 or option2.

optional arguments are in []
exclusive or is represented by |
--]]
```

- `num_zooms`

```
--[[
Zs = num_zooms()

ret: Zs, number of zoom positions
--]]
```

Get the number of system zooms.

- `set_zooms`

```
--[[
set_zooms(Zs)

Zs: number of zoom positions
--]]
set_zooms(3) -- 3 configuration system.
```

Sets the number of zoom positions for the system.

Maximum number of zooms: 25.

- `ins_zoom`

```
--[[
ins_zoom(at_pos)

at_pos: Zoom position number at which new zoom position is
inserted.
--]]
ins_zoom(3) -- Insert zoom at position 3.
```

Inserts a system zoom position at the designated position.

- `del_zoom`

```
--[[
del_zoom(pos)

pos: position number to be deleted
--]]
del_zoom(3) -- Delete third zoom psoition.
```

Delete the system zoom position at the designated position.

- `sys_dezoom`

```
--[[
sys_dezoom(to_pos)

to_pos: position number to dezoom to
--]]
sys_dezoom(3) -- Dezoom system to position 3.
              -- After this, system is unzoomed.
```

Deletes all zoom position but the designated one.

- `copy_zoom`

```
--[[
copy_zoom(to_pos, from_pos)

to_pos: destination position
from_pos: source position
--]]
```

Copies parameter values from the source position to the destination position.

- `list_zooms`

```
--[[
list_zooms()

lists the zoomed parameters. An example listing is shown:

 Zoom/Prm       1       2       3
      CUY S 3 : 0.006741        0.010741        0.008741
      THI S 2 : 5       4       2
--]]
```

- `display_zoom`

```
--[[
display_zoom(pos)

pos: zoom position number to use as default when listing data and
    displaying results. Default: 1
--]]
```

4.1.2 Reactors

The next kind of parameters are the reactors. As the name suggests, they react to changes in another "source" parameters of a similar type. In the simplest form, they will just follow the source and have the same value as the source. But they can also support scales and offset with respect to the source. Parameters that are reacting have a letter "r" next to their listing. The functionality of zoomers is a subset of the functionality of reactors.

- `list_reacts`

```
--[[
list_reacts()

The listing has two columns, first is destination, other is source.
```

```
Example listing:

    THI S 4:  z 1 <---  z 1 ___pim_dist__ [1, 0]

In the above listing, thi s 4 of the first zoom position is
reacting to the paraxial image distance calculation of the
first zoom with a scale factor of unity and offset of zero.
--]]
```

4.1.3 Variators

These are reactors with additional properties. As the name suggests, these are numeric parameters and can be designated as variables for optimization. Tolerances can also be assigned to them and `touch()` function will perturb these parameters based on the tolerance. The functionality of reactors is a subset of the functionality of variators. When a parameter is a variable, the letter "v" is displayed next to its listing. The following are the system level function associated with variators.

- `list_vars`

```
--[[
list_vars()

Example listing:
           ID : Zs
      CUY S 2 : 1
      THI S 3 : 2

curvature of S 2 for zoom 1 is variable. Thickness of S 3 for zoom
   2 is also a variable.
--]]
```

- `list_tols`

```
--[[
list_tols()
```

```
Example listing:
           ID : Zs
      THI S 3 : 0.025

Thickness of S 3 should tolerate an error of 25 microns. Tolerances
    are listed for every zoom if the parameter is zoomed.
--]]
```

- `touch`

```
--[[
touch()

Perturb the system parameters that have tolerances defined.
   Currently, the perturbation amount is uniformly distributed
   between -tol and +tol.
--]]
```

- `scale_tols`

```
--[[
scale_tols(factor)

factor: The scaling multiplicative factor is applied to all
   tolerances.

Note, instead of using a scale of 0.0 to deactivate a tolerance,
   delete the tolerance directly.
--]]
```

4.1.4 Parameter Functions

The following is a list of functions that are directly or indirectly associated with parametric actions. The different parameters associated with a particular aspect will be discussed in the following chapters.

- del

```
--[[
del(id)

id: String name of item to delete.

Deletes the item identified by the id.
--]]
del("W 2") -- Deletes wavelength 2.
```

- num_of

```
--[[
N = num_of(id)

N: Number of items of type identified by id.
id: String identifying the items, like "S", "W" etc.
--]]
CPH> print("Seq-surfaces count: " .. num_of("s"))
Seq-surfaces count: 3
```

- set_label

```
--[[
set_label(id, label)

id: Item identifier.
label: String label for easier addressing.

Labels are case sensitive.
--]]
CPH> set_label("s 2", "Stp") -- Stop label
CPH> set("thi s 'Stp'", 1.234) -- Directly address stop
```

- get_label

```
--[[
label = get_label(id)

label: Returns the string label of the item.
id: Identifying string for the item.

Empty string returned if no label defined.
--]]
```

- `ordinal`

```
--[[
pos = ordinal(label)

pos: Numeric position of the item in the list.
label: Label string of the item.
--]]
CPH> local pos = ordinal("s 'Stp'") -- 2
```

It returns the numerical position of the labeled item.

- `set`

```
--[[
set(id, val, [zm])

id: Parameter id string.
val: Value (number, string, bool, table).
zm: Numeric zoom position to target, default: 1.
--]]
CPH> set("thi s 2", 1.0, 1)
CPH> set("gls2 s 2", "n-bk7|schott")
CPH> set("use_jm s 2", true) -- Use Jones Matrix flag
CPH> set("apdz s 2", {0, 0, 1.0, 1.0}) -- Appodization
```

Function to set parameter values.

- `get`

```
--[[
val = get(id, [zm])

val: Returns the value (number, string, bool, table).
id: Parameter id string.
zm: Numeric zoom position to target, default: 1.
--]]
CPH> val = get("apdz s 2")
CPH> list_tbl(val)
{0, 0, 1, 1}
```

Function returns the value of the targeted parameters.

- zoom

```
--[[
zoom(id)

id: Parameter id string.

Zooms the parameter, allowing it to represent different values for
    different zoom positions.
--]]
```

- dezoom

```
--[[
dezoom(id, [to_zm])

id: Parameter id string.
to_zm: Dezoom to position number, default: 1.
--]]
```

- is_zoomed

```
--[[
bool = is_zoomed(id)

bool: Returns true of parameter is zoomed.
```

```
id: Parameter id string.
--]]
```

- `set_reaction`

```
--[[
set_reaction(r, s, [scale], [offset], [r_zm], [s_zm])

r: Reacting parameter id string.
s: Source parameter id string.
scale: Default: 1.0.
offset: Default: 0.0.
r_zm: Zoom number of reacting parameter value, default: 1.
s_zm: Zoom number of the source value.

However, scale and offset are interpreted according to the reacting
    parameter type. The reaction is as follows:
numeric types  : r_value = s_value * scale + offset
booleans types : r_value = s_value,  if scale >= 1.0
               : r_value = !s_value, if scale <  1.0
               : examples are ignr, use_jm, etc.
other types    : r_value = s_value
               : examples are gls2, flm, etc.
--]]
CPH> set_reaction("thi s 2", "thi s 1", 1.0, 0.0, 1, 1)
```

Sets up reaction connecting source parameter to reacting parameter which then reacts to changes in the source. Reactions cannot be setup between differing types, that is for example between strings (glass) and numbers (thickness). When the parameter type is boolean, the following convention is recommended, `scale = 1.0 => r_value = s_value; scale < 1.0 => r_value = !s_value`.

Maximum chain reaction length: 100.

Cyclic reaction not allowed.

- `del_reaction`

```
--[[
del_reaction(id, [zm])

id: Reacting parameter id string.
zm: Zoom number of the reacting value, default: 1.

Delets the reaction connecting source to reacting parameter.
--]]
```

- is_reacting

```
--[[
bool = is_reacting(id, [zm])

bool: Returns true if reacting.
id: Parameter id string.
zm: Zoom number of the value, default: 1.
--]]
```

- reaction_source

```
--[[
bool = reaction_source(id)

bool: Returns true if parameter is source of reaction for any zoom.
id: Parameter id string.
--]]
```

- vary

```
--[[
vary(id, [zm])

id: Parameter id string.
zm: Zoom number, default: 1.

Designates the parameter value at the specified zoom to be a
   variable during optimization. Listings will indicate this by the
    letter "v" next to the parameter.
```

```
--]]
```

- `freeze`

```
--[[
freeze(id, [zm])

id: Parameter id string.
zm: Zoom number, default: 1.

Freezes the parameter value at the specified zoom, that is it is
no longer a variable during optimization.

Without any argument, the function freeze() freezes the system.
--]]
```

- `is_variable`

```
--[[
bool = is_variable(id, [zm])

bool: Returns true of value is variable.
id: Parameter id string.
zm: Zoom number, default: 1.
--]]
```

- `set_tol`

```
--[[
set_tol(id, val, [zm])

id: Parameter id string.
val: Tolerance value (numeric).
zm: Zoom number, default: 1.

Sets the tolerance on a parameter.
--]]
```

- del_tol

```
--[[
del_tol(id, [zm])

id: Parameter id string.
zm: Zoom number, default: 1.

Deletes the tolerance on a parameter.

Without any argument, the function del_tol() deletes all
tolerances.
--]]
```

- get_tol

```
--[[
val = get_tol(id, [zm])

val: Returns tolerance value.
id: Parameter id string.
zm: Zoom number, default: 1.
--]]
```

- is_tolerating

```
--[[
bool = is_tolerating(id, [zm])

bool: Returns true of parameter has tolerance setup.
id: Parameter id string.
zm: Zoom number, default: 1.
--]]
```

- list

```
--[[
list(id)
```

```
id: Id string.

Print listing of the numbered item. Or a summary of all items if
   position number omitted.
--]]
CPH> list"w 1"
W 1
       WL W 1 : 0.5
       WT W 1 : 1
CPH> list"w"
1 W entrie(s).

W 1
       WL W 1 : 0.5
       WT W 1 : 1

PWL     1

          TEM : 20
          PRE : 1
           RH : 0
```

4.1.5 Alternative Compact Interface

As mentioned earlier, Lua is highly customizable. An alternative interface is available to manipulate the parameters[2]. Use of this interface is shown in the following example script.

```
-- Setting
s[2].thi = 1.23 -- Set thi on s 2 to 1.23. Zoom 1 is assumed.
z[1].s[2].thi = 1.23 -- same as above.
s[3].thi = s[1].thi + s[2].thi
s[4].repos = "bend" -- Repositioning mode is now "bend"
-- Getting
local rad = z[2].s[3].rdy -- Get zoom 2 rdy on s 3.
```

[2]This interface will be further improved upon. Currently only `set` and `get` functions are emulated. A few parameters that are not associated with numbered items cannot be set in this manner. Thin film parameters are also not addressed by this approach. Labels cannot be set, but can be used instead of numbers.

```
local ignore_flag = s[2].ignr
set_label("s 2", "stp")
local stp_sz = s['stp'].aut_ape
```

To realize this interface, the following tables have been placed in the Lua global environment, `s, n, w, f, z`. The order of table access in the above examples is important.

4.2 Utilities

These are function that are used throughout the application.

4.2.1 General Utility Functions

- `help`

```
--[[
help()

Launches this reference manual.
--]]
```

- `exit`

```
--[[
exit

exit command exits the application.
--]]
```

- `get_cores`

```
--[[
c = get_cores()

c: Returns the number of CPU cores being utilized.
--]]
```

- `set_cores`

```
--[[
set_cores(c)

c: Number of CPU cores to utilized.

c = 0 => auto detect number of logical CPUs.
--]]
```

Note that setting the cores to a number larger than the supported number of cores is not helpful additionally this follows the law of diminishing returns. Default is to auto detect and this is probably acceptable in most cases.

- `list_all`

```
--[[
list_all(dsp)

dsp: Boolean flag. False => simplified listing. Full otherwise.
--]]
```

- `set_numeric_precision`

```
--[[
set_numeric_precision(digits)

digits: Number of decimal digits to list.

Set the numerical precission of the text output.
--]]
```

- `tbl2str`

```
--[[
str = tbl2str(tbl, [indent])

str: Returns string representing the table.
tbl: Table to be printed.
```

```
indent: Number representing starting indentation for readability,
    default: 0

Lua can have nested tables and also cyclic tables. Please note that
     cyclic tables are not supported by this function. For nested
    tables, indentation is used for better readability. This
    function is not used much by itself.
--]]
```

- list_tbl

```
--[[
list_tbl(tbl)

tbl: Table to be listed with proper indentations.

It calls tbl2str and prints the results.
--]]
CPH> tbl = {a = 1, b = {c = 3, d = 4}}
CPH> list_tbl(tbl)
{
 a = 1,
 b = {
  c = 3,
  d = 4
 }
}
```

- general_equality_test

```
--[[
res = general_equality_test(a, b)

res: Returns boolean, true if a == b.
a, b: Values or tables
--]]
```

- `check_globals`

```
--[[
check_globals(v)

v: bool

Sets the check globals flag, like so
check_globals(true)
--]]
```

With the check globals flag set to true, attempt to access the value of an undefined variable results in an error. Further more, defining new global variable in the Lua global environment is also prohibited. However, it is still possible to declare new global variables in the users global environment. Default value of this flag is false.

4.2.2 Data Display Functions

- `plot`

```
--[[
plot(data, [plot_args])

data: data is a table of subtables.
      Each subtable is a data point, that is {x, y1, y2, ...}.
      if #subtable = 1, then x = 1, 2, 3, ... .
plot_args = {
    [type] = "line",   --Type of plot: scatter|line|bar.
    [labels] = {"Data_1", "Data_2", ...},
    -- Labels for data, size must match data.
    [title] = "",      -- Title of the plot.
    [subtitle] = "",   -- Subtitle of the plot.
    [xlabel] = "",     -- X-axis label.
    [ylabel] = "",     -- Y-axis label.
    [xmin] = auto,     -- Minimum x-axis value.
    [xmax] = auto,     -- Maximum x-axis value.
    [ymin] = auto,     -- Minimum y-axis value.
    [ymax] = auto,     -- Maximum y-axis value.
}
```

```
--]]
-- Contents of test.lua:
local data = {
    {1, 2, 3},
    {2, 2.1, 3.2},
    {3, 2.2, 3.4},
    {4, 2.3, 3.6},
    {5, 2.4, 3.8},
    {6, 2.5, 4.0},
    {7, 2.6, 4.2},
    {8, 2.7, 4.4},
    {9, 2.8, 4.6}
}

local plot_args = {
    title = "Test Plot",
    labels = {"Y1", "Y2"},
    xlabel = "X",
    ylabel = "Y"
}

plot(data, plot_args)

-- Run test.lua:
CPH> runfile"test"
```

Result of the above run is shown in figure 4.1. The charts are interactive, the chart labels can be (un)selected by clicking. This can help de-clutter dense plots.

- `hist_table`

```
--[[
d = hist_table(data, [bins])

d: Returns histogram table of data distributed in bins
data: Table of data.
bins: Number of histogram bins, default: 10.
--]]
-- Contents of test.lua:
```

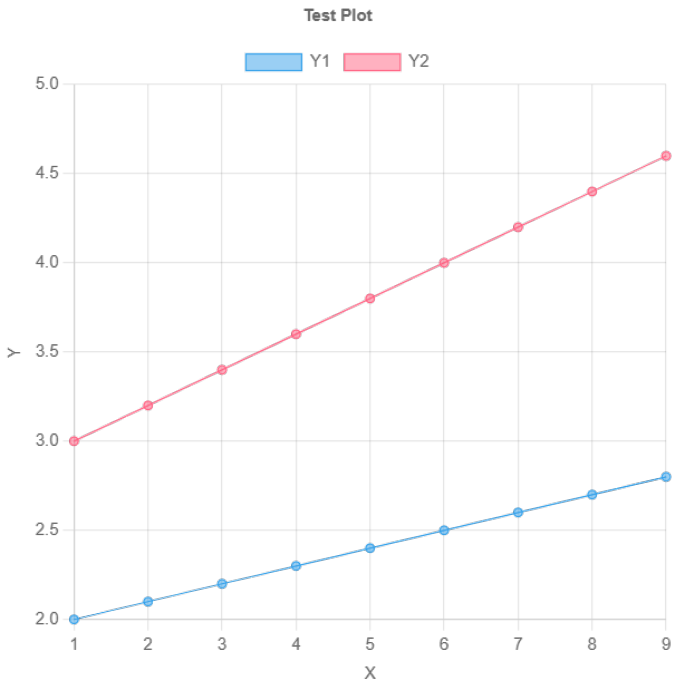

Figure 4.1: Example plot.

```
-- Generate a array of 100 numbers
-- randomly distributed between 0 and 10
local array = {}
for i = 1, 100 do
    array[i] = math.random() * 10
end

local d = hist_table(array)
plot(d, {type="bar"})

-- Run test.lua
CPH> runfile"test"
```

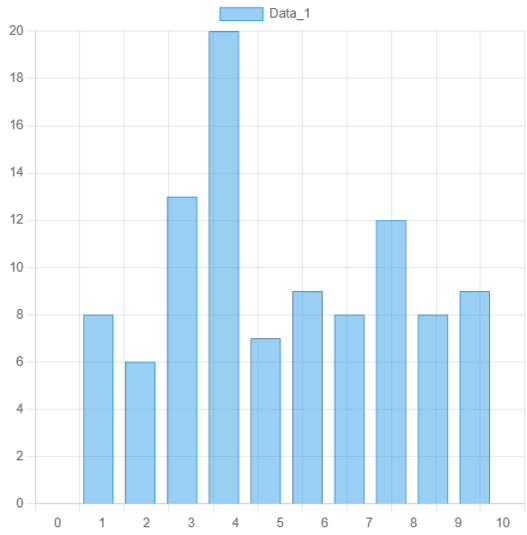

Figure 4.2: Example histogram plot.

Result of the above run is shown in figure 4.2.

- `showlens`

```
--[[
showlens([args])

args = {
   [sym] = "YZ",    -- Symmetry to use, YZ|XZ|NONE,
```

```
                    -- NONE draws surfaces and YZ, XZ sections.
   [overlay] = "S 1",   -- Overlay surface id.
                        -- "S 1" => no overlay, show selected zoom.
                        -- Overlay frame taken from first zoom.
   [rays] = {
     [f] = "f <f>"|"all",  -- Default is "all" for all fields.
                           -- This does not apply to user rays.
     [w] = "w <w>"|"all"|"pwl"
                           -- Default is "pwl" for the zoom.
   }
}
--]]
-- Contents of test.lua:
local args = {
   sym = "none",
   rays = {
      f = "all",
      w = "pwl"
   }
}
showlens(args)

-- Run test.lua
CPH> runfile"test"
```

Result of the above run is shown in figure 4.3. The function `showlens(args)` internally calls `get_scene(args)` to obtain a table of positional information about the optical system. This table is then passed to the function `display (scene, args)`, which displays it in the browser.

The display is always 3D even when only the YZ symmetry is selected. Click + drag results in reorientation, shift + click + drag results in panning, scroll wheel is for zooming in and out. Right click exposes a menu item to reset the display.[3]

[3]Export is currently limited to STL files. These have limited functionality but are still useful for checking against packaging requirements in a traditional CAD software.

When using external browser for display, it can be necessary to pause between multiple display commands in a script to allow display functions to catch up. There is no inbuilt function to pause in Lua. However we can achieve this by calling the appropriate system function. For example, on a Windows machine, `os.execute("Sleep 1")` will pause for one second.

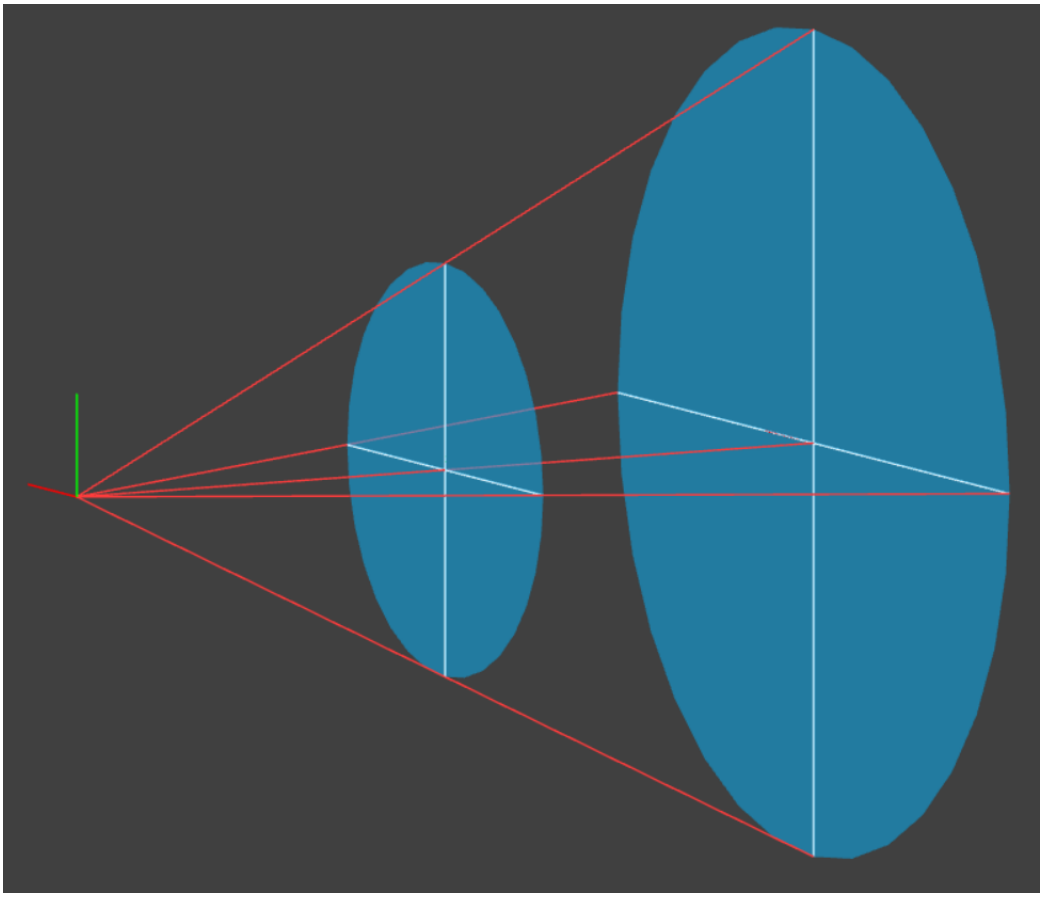

Figure 4.3: Output from `showlens`.

- list_gvr

```
--[[
list_gvr()

Lists the global variables defined.
--]]
CPH> num_val = 3.141
CPH> str_val = "a string"
CPH> function doodle() print("hello") end
CPH> tbl = {1, 2, 3}
CPH> opt = Solver.new()
CPH> list_gvr()
num_val                                 3.141
str_val                                 "a string"
tbl                                     table
doodle                                  function
opt                                     solver
```

4.2.3 File System Functions

- cd

```
--[[
cd(path)

path: String representing path to a folder.

Changes the working folder to the one represented by path.
Example:
cd("C:/Users/MyName/Documents/project")
--]]
```

- pwd

```
--[[
pwd()

Prints the current working folder path.
```

```
--]]
```

- ls

```
--[[
ls()

Lists the sorted contents of the current folder.
Directories are designated by (D).
--]]
```

- add_path

```
--[[
add_path(path)

path: String, path to folder to add to paths list.

Files are searched, first in the working folder,
then in the folders in the paths list,
in the order of the list.
--]]
```

- del_path

```
--[[
del_path(path)

path: String, path to be removed.
--]]
```

- list_paths

```
--[[
list_paths()

Lists the paths in order.
--]]
```

- `find_a_path`

```
--[[
pth = find_a_path(file)

pth: Returns string representing full path to file.
file: String, name of file, extension optional.

Returns full path to file.
Checks for existence, returns "" if not found.
Mainly for loading/running files.
--]]
```

- `qualify_filename`

```
--[[
pth = qualify_filename(file)

pth: Returns string representing full path to file.
file: String, file name, extension optional.

Qualify filename with full path and default extension.
Does not check for file existence. Mainly for saving files.
--]]
```

- `runfile`

```
--[[
ret = runfile(file)

ret: Captures anything that the script file returns.
file: String, scipt file to run, extension optional.

Executes the commands in the script file.
A return statement can only occur at the end of a code block, like
a function. The script in the file is a code block and like
functions, a return statement can be placed at the end of the
script.
--]]
```

The return from `runfile` is useful too. For example, the script file can define local functions and return a table of these functions. All the details of the implementation of these functions is encapsulated in the script.

- `savefile`

```
--[[
savefile(file),

file: String name of file to save to.

Saves the optical system to script file. Extension is optional.
Scripts for any films and user defined glasses used by the system
are also generated and saved in the file.
--]]
```

- `editfile`

```
--[[
editfile(file)

file: String name of file to edit.

Opens the file in the default editor for the file.
--]]
```

4.2.4 Undo

- `undo`
- `redo`
- `pause_undo`
- `unpause_undo`
- `list_undostack`

```
--[[
undo()              -- Undo last action.
```

```
redo()              -- Redo last action.
pause_undo()        -- Pauses the undo feature.
unpause_undo()      -- Restart the unfo feature.
list_undostack()    -- Lists the undo stack.
                    -- <--- marks the stack pointer.
--]]
CPH> list_undostack()
Undo Stack Listing:
 set("thi s 2", 3.141)
 ins_surf("s 2")
 set_stop(2)    <---
```

Probably the most used function here is `undo()`. Furthermore, its main use case is to undo the actions of an optimization cycle, and the `touch` function.

Maximum stack depth: 13.

4.2.5 Lens Utilities

- `blank`

```
--[[
blank()

Reset the entire model reulting in a blank lens.
--]]
```

After this function the system sate is as follows. All non-sequential surfaces are deleted. There are three default sequential surfaces one mm apart from previous surface. There is a single field with object angle of zero degrees. Pupil type is EPD with value of 1 mm. There is single wavelength ($0.5\mu m$) and temperature is 20.0 degrees centigrade, pressure is 1.0 atmosphere and relative humidity is 0%.

Polarization coordinate system is collimated and polarization state is horizontal ($E_x = 1; E_y = 0$), and the YZ plans is used for reference ray calculations.

The film store is emptied and user defined glasses are purged from the glass house. Also, the system has a single zoom position defined. The internal cost function parameters are also reset to defaults.

- `scale_lens`

```
--[[
scale_lens(scale_factor, [input])

scale_factor: The scale factor cannot be zero.
input = {
  [system] = true,
  [range] = {
    "s <start>",
    "s <stop>"
  }
}

Scales the dimensional parameters of the whole system or selected
surfaces. With system = true, system data like EPD, field heights
are scalled. In the absence of range, all surfaces (sequential
and non-sequential) are scaled. When a valid range is provided,
default value of system is false.
--]]
```

Please note:

- Field values are scaled only when field type (`FTYP`) for all zooms is set to `"HEIGHT"`.
- The thin film thickness taper parameter (`TFD`) is not scaled.
- Surface apodization parameter (`APDZ`) is not scaled.

4.3 System Settings

Defining the number of zoom positions for the system is done by function `set_zooms`. System focal mode is controlled by the parameter `FOCAL_MODE`.

4.3.1 Fields

- `ins_fld`

```
--[[
ins_fld(num, [data])

num: Field number.
data = {
  [x]   = 0.0, -- x value
  [y]   = 0.0,
  [vxp] = 0.0, -- Vignetting factor, +x
  [vyp] = 0.0,
  [vxm] = 0.0,
  [vym] = 0.0
}

Each parameter of the field can be addressed directly as well.
--]]
```

Maximum number of fields: 25.

4.3.2 Wavelengths

- `ins_wl`

```
--[[
ins_wl(num, w)

num: Wavelength number.
w: Wavelength in micro meters.
--]]
```

Maximum number of wavelengths allowed: 25.

Minimum wavelength allowed: 0.001 microns.

- `set_pwl`

```
--[[
set_pwl(wl, [zm])

wl: Wavelength number to use.
```

```
zm: Zoom number, default: 1.

Set primary wavelength for the zoom.
--]]
```

- get_pwl

```
--[[
wl = get_pwl([zm])

wl: Returns the PWL number.
zm: Zoom number, default: 1.
--]]
```

- zoom_pwl

```
--[[
zoom_pwl()
--]]
```

- dezoom_pwl

```
--[[
dezoom_pwl([zm])

zm: Zoom number, default: 1.
--]]
```

- is_pwl_zoomed

```
--[[
bool = is_pwl_zoomed()

bool: Returns true of PWL is zoomed.
--]]
```

4.3.3 Reference Rays

- `pim_reaction`

```
--[[
pim_reaction(bool, [zm])

bool: True to set the reaction, false to deactivate.
zm: Zoom number, default: 1.

Sets a reaction on thickness of surface before image to place image
at ideal image location.
--]]
```

- `iterate_all_chiefs`

```
--[[
iterate_all_chiefs(bool)

bool: True to iterate all chief rays.

Iterate all chief rays to go through the stop center.
Otherwise, only iterates F 1 PWL chief rays.
--]]
```

- `iterate_marginals`

```
--[[
iterate_marginals(bool)

bool: True to iterate all marginal rays to stop edge.

Targets marginal rays to edge of first order stop, ignoring
vignetting factors.
--]]
```

- `list_fo`

```
--[[
list_fo([zm])

zm: Zoom number, default: 1.

Lists the first order properties for zoom.
--]]
CPH> list_fo()
First order properties (ray and surface local) for zoom 1:
FO:
        F1 = -1e+13
        F2 = 1e+13
     m_ENP = 1
      m_XP = 1
   ENP_POS = 0
       PP1 = 0
       NP1 = 0
    XP_POS = 0
       PP2 = 0
       NP2 = 0

   Ref ENP = 1
   ENP POS = 0, 0, 0
   ENP DIR = 0, 0, 1
    XP POS = 0, 0, 0
    XP DIR = 0, 0, 1
```

- ref_status

```
--[[
ref_status()

Lists any error messages for reference rays.
--]]
CPH> ref_status() -- All good here.
F1 PWL ray errors:

Remaining rays errors:
Z01:
Diagnostic errors:
```

4.4 Glasses

- `add_myglass`

```
--[[
add_myglass(info, data)

info = {
    glass = <Glass_Name>,
    [density] = 0, -- grams per cubic centimeter
    [cost] = 0.0, -- relative cost, relative to SCHOTT N-BK7.
                  -- >= 0. 0 => cost not available.
    [homogeneous] = true,
    [type] = "SOLID" -- SOLID|LIQUID|GAS|UNIAXIAL
}
data = {
    [cte] = "QUADRATIC",
    [cte_data] = {c0, c1, c2, t0},
        --       {0, 0, 0, 20.0} (defaults)
        -- Empty table is auto interpreted according to CTE.
        -- Omega(tem) =
        --  (c0*tem + c1*pow((tem-T0), 2)/2
        --  + c2*pow((tem-T0), 3)/3);
        -- L2 = L1 * exp(Omega(T2) - Omega(T1))
        -- 1e-6 common factor assumed.

    [alpha] = "ORDINARY",   -- ORDINARY|INTERPOLATE,
                            -- ORDINARY is calculated from index.
    [alpha_data] = { {} },  -- Empty table is interpreted according
                            -- to alpha.
                            -- = { {ws}, {Ts} } for INTERPOLATE
                            -- internal transmission data.
                            -- first w is ignored,
                            -- first T is slab thickness.

    [dndt] = "SCHOTT",      -- GAS|SCHOTT
    [dndt_data] = {},       -- Empty table is auto interpreted
                            -- according to dndt type.
        -- SCHOTT ->
        --   {d0, d1, d2, e0, e1, lambdaTK, ref_tem, mes_tem}
```

```
        --   {0, 0, 0, 0, 0, 0, 20.0, 20.0} (defaults)
        -- GAS ->
        --   {a, b, c, d, e, f, T_m, lower_wl, upper_wl}
        --   {6432.8, 2949810.0, 146.0, 25540.0, 41.0, 0.0034785,
        --    15, 0.17, 20.0} (defaults)

    [dispersion] = "INTERPOLATE", -- GAS|INTERPOLATE|SELLMEIER
    [dispersion_data] = {}        -- Empty table is interpreted as:
        -- GAS -> as above.
        -- INTERPOLATE -> { {ws}, {ns}, [{ks}] }, no duplicate
        -- wavelengths allowed. Data not optional.
        -- Absorption => imaginary part of index > 0.
        -- SELLMEIER -> {B1, B2, B3, C1, C2, C3,lower_wl,upper_wl},
        -- data not optional.

        -- Note that these tables are numeric sequences in the
        -- above order. Do not use keys.
}
--]]
-- Example script to define a user glass:
local info = {glass = "gsl1"}
local data = {
    dispersion = "interpolate",
    dispersion_data = {
        {0.4,   0.5,     0.6,    0.7},  -- wavelengths
        {1.6,   1.51,    1.4,    1.3}   -- n
    }
}

add_myglass(info, data)
```

The index of refraction of gas and that of air is based on formula given in book by F. Kohlrausch[7]. Note that this formula produces similar results as that of the Cauchy's formula[1] at the right temperature and pressure. Schott DNDT is based on a Schott technical article[9].[4]

For interpolate dispersion, maximum wavelength entries: 100.

Maximum number of user glasses: 100.

[4]UNIAXIAL and GRIN not yet implemented

- del_myglass

```
--[[
del_myglass(name)

name: String, user glass name.

Deletes the user defined glass.
--]]
```

- list_glass_info

```
--[[
list_glass_info(name)

name: String, glass name.

Lists the infromation about the glass.
--]]
```

- get_glass_info

```
--[[
tbl = get_glass_info(name)

tbl: Returns a table containing glass infomation.
name: String, glass name.

example return = {
   name      = "Glass_Name",
   density   = 0,
   rcost     = 1.0,
   hom       = true,
   state     = "SOLID"
}
--]]
```

- `list_myglasses`

```
--[[
list_myglasses()

List all user defined glasses.
--]]
```

- `add_catalog_glass`

```
--[[
bool = add_catalog_glass(info, data, cat_name)

bool: Returns true on success.
info: Glass info table.
data: Glass data table.
cat_name: String, catalog name.
--]]
```

- `add_glass_catalog`

```
--[[
add_glass_catalog(vendor)

vendor: String, name of the glass vendor, like Schott.

Loads the entire catalog into memory. Catalog has to be first
loaded then its glasses can be used in the model.
--]]
```

Maximum catalog glasses in memory: 1000.

- `remove_glass_catalog`

```
--[[
remove_glass_catalog(vendor)

vendor: String, name of the glass vendor, like Schott.
```

```
Unloads the entire catalog from the memory.
--]]
```

- list_loaded_glasses

```
--[[
list_loaded_glasses()

Lists all the vendor glasses in the memory.
--]]
```

- air_index

```
--[[
n = air_index(wl, [zm])

n: Returns the index of refraction of air.
wl: Wavelength number.
zm: Zoom number, default: 1.
--]]
```

- get_medium

```
--[[
med = get_medium()

med: Returns string name of the default medium like AIR.
--]]
```

4.5 Films

- add_film

```
--[[
add_film(tbl)
```

```
tbl = {
    film = <name>,
    layers = {
        {<glass>, <thickness>}...
    }
}

Light usually incident from top, that is the first layer.
The last layer attaches to the substrate.
Thickness is in nano meters.
--]]
-- Example script to add a film stack.
local stack = {
    film = "my_film",
    layers = {
        {"gsl1", 110},
        {"gsl2", 120}
    }
}

add_film(stack)
```

Maximum number of layers: 100.

Maximum number of stacks: 20.

- `del_film`

```
--[[
del_film(film)

film: String name of the thin film.
--]]
```

- `list_films`

```
--[[
list_films()
```

```
List all the film stacks defined.
--]]
```

4.6 Surfaces

4.6.1 Surface Related

- `ins_surf`

```
--[[
ins_surf(id, [type|data])

id: String, surface id.
type: String, surface type, default: SPHERE.
data = {
   [type]     = "SPHERE",
   [glass]    = <medium>, -- For GLS2.
   [rdy]      = 0,
   [thi]      = 0
}

Supported surface types = SPHERE|CONIC|QCON|TORIC|NSIN|NSOUT
NSIN|NSOUT cannot be inserted in the non-sequential surface list.
--]]
```

Maximum SEQ surfaces: 100.

Maximum NSEQ surfaces: 100.

- `change_surf`

```
--[[
change_surf(id, type)

id: String, surface id, like "s 2".
type: String, surface type, like "conic".
--]]
```

- `profile_data`

```
--[[
res = profile_data(surf_id, x, y, [zm])

res: Returns table = {
        {sag, sag_x, sag_y},
        {sag_xx, sag_yy, sag_xy},
        {normalX, normalY, normalZ}
     }
     On error, empty.
surf_id: String, surface id.
x, y: X and Y coordinate in surface local frame.
zm: Zoom number, default: 1.
Here sag_x is derivative of sag wrt x, sag_xx is the second
derivative etc.

Returns the surface sag, normal at [x, y], and derivatives of sag.
Normal points into GLS1, that is towards -Z.
--]]
```

- `indexo`

```
--[[
n, k = indexo(glass_id, wl, [zm])

n, k: Returns two numbers, n and k.
glass_id: String, like "GLS1 n 2".
wl: Wavelength number.
zm: Zoom number, default: 1.

Get ordinary absolute index of refraction.

For sequential surfaces, just use surface id like "s 1",
GLS1 and GLS2 is ignored.
Returns index of the following medium for the ray.
--]]
```

- omega

```
--[[
omg = omega(glass_id, zm)

omg: Returns the number omega for the glass.
glass_id: String, e.g. "GLS1 n 2".
zm: Zoom number, not optional.

The thermal expansion is: L2 = L1 * exp(omega_2 - omega_1),
where omega_1 and omega_2 are the omegas of the glass at the two
temperatures. For sequential surfaces, always returns GLS2 omega.

Temperature differential comes by zooming the system temperature.
--]]
```

4.6.2 Stop Related

- set_stop

```
--[[
set_stop(surf, [zm])

surf: Sequential surface number.
zm: Zoom number, default: 1.

Designates surf to be the system stop for this zoom.
surf cannot be the firt (object) or the last (image) surface.
--]]
```

- get_stop

```
--[[
stp = get_stop([zm])

stp: Returns the sequential surface number that is stop.
zm: Zoom number, default: 1.
--]]
```

- zoom_stop

```
--[[
zoom_stop()

Zooms the spot pointer.
--]]
```

- is_stop_zoomed

```
--[[
bool = is_stop_zoomed()

bool: Returns true if stop pointer zoomed.
--]]
```

- dezoom_stop

```
--[[
dezoom_stop([to_zm])

to_zm: Zoom number to collapse to, default: 1.
--]]
```

4.6.3 Global Referencing

- set_glb_ref

```
--[[

set_glb_ref(surf_id, ref_id)

surf_id: String, surface id.
ref_id: String, reference surface id.

surf_id is placed with respect to ref_id.

Within the list, only backward references are allowed.
```

```
And sequential surfaces cannot refer to non-sequential surfaces.
--]]
```

- `del_glb_ref`

```
--[[
del_glb_ref(surf_id)

surf_id: String, surface id.

Delete a global positional reference.
--]]
```

- `list_glb_refs`

```
--[[
list_glb_refs()

Lists a table showing the global references linkage.
--]]
```

- `list_glb_position`

```
--[[
list_glb_position(surf_id, [zm])

surf_id: String, surface id.
zm: Zoom number, default: 1.

Prints the global position of a surface.
--]]
```

4.7 Rays

It is assumed that all rays start in isotropic and homogeneous medium. Node power = E^2, and index of refraction automatically accounted for. Reference rays and sequential analysis rays have unit starting power. Maximum number

of user defined rays per wavelength and zoom is limited to 169. If object is too far, reference rays and sequential analysis rays will start on "S 2". If the ratio of entrance pupil radius to the object distance is less than about 2×10^{-5}, object is then assumed to be too far.

4.7.1 User Rays

- add_ray

```
--[[
add_ray(node_tbl)

node_tbl: Table containing ray starting information.

node_tbl = {
    pos   = {x, y},
    [dir] = {0, 0}, -- Tanx and Tany.
    [jv]  = { {1, 0},
              {0, 0} },
            -- Ex', Ey'
    [sur] = "S 1",
    [seq] = true,
    [w]   = <PWL>, -- Wavelength number.
    [z]   = 1
}

For non-sequential user rays, please ensure that they do not miss
   the starting surface.
--]]
```

- del_ray

```
--[[
del_ray(ray_num, wl, [zm])

ray_num: Ray number.
wl: Wavelength number.
[zm]: Zoom number, default: 1.
--]]
```

- `reset_user_rays`

```
--[[
reset_user_rays()

Remove all user defined rays.
--]]
```

- `num_of_user_rays`

```
--[[
num = num_of_user_rays(wl, [zm])

num: Returns number of user defined rays.
wl: Wavelength number.
zm: Zoom number, default: 1.
--]]
```

4.7.2 Ray Information

- `ray_endinfo`

```
--[[
res = ray_endinfo([rid])

res: Results table.
rid: Ray identification table.

Example rid = {
    [f] = 1,    -- Field. Use 0 for user defined rays.
    [w] = <PWL>,-- Wavelength number.
    [z] = 1,
    [r] = 0|"ch" or 1 -- Ray number. Defaults to 1 for user rays
                      -- and 0 for reference rays.
                      -- See reference ray designation below.
}

-- Reference ray designation:
-- rnum    Ref Ray         (string)
```

```
-- 0    -> Chief ray       (CH).
-- 1    -> +X marginal ray (XP).
-- 2    -> -X marginal ray (XM).
-- 3    -> +Y marginal ray (YP).
-- 4    -> -Y marginal ray (YM).

Example return = {
    w = <wl>,
    z = <zm>,
    seq = <bool>,
    [error] = <string>, -- If there was an error.
    node = {
        {
            pos = {x, y, z},
            dir = {l, m, n},
            E = {Exr, Exi, Eyr, Eyi, 0, 0},
            sur = <str>,   -- Info is local to this surface.
            type = <str>,  -- Incident, T, R, etc.
            medium = <str>,-- Glass name.
            phase = <phs>, -- Accumulated geometric phase.
            ncost = {r, i} -- Index * obliquity factor.
        }
    }
}

A ray segment is bounded on each side by a ray node.
--]]
CPH> res = ray_endinfo()
CPH> list_tbl(res)
{
 node = {
  {
   dir = {0, 0, 1},
   pos = {0, 0, 0},
   type = "I",
   phase = 2.001,
   sur = "S 3",
   ncost = {1, 0},
   medium = "AIR",
   E = {0.9999, 0, 0, 0, 0, 0}
```

```
  }
 },
 z = 1,
 w = 1,
 seq = true
}
```

- ray_info

```
--[[
res = ray_info([rid])

res: Results table.
rid: Ray identification table.

Return all nodes in the ray.
Example return = {
    w = <wl>,
    z = <zm>,
    seq = <bool>,
    nodes = { -- Array of nodes.
        { -- Node 1.
            pos = {x, y, z},
            dir = {l, m, n},
            E = {Exr, Exi, Eyr, Eyi, 0, 0},
            sur = <str>,   -- Info is local to this surface.
            type = <str>,  -- Incident, T, R, etc.
            medium = <str>,-- Glass name.
            phase = <phs>, -- Accumulated geometric phase.
            ncost = {r, i} -- Index * obliquity factor.
        } ...
    }
}
--]]
```

- ray_opd

```
--[[
opd = ray_opd(rid, surf_up, surf_down)
```

```
opd: Return the optical path difference.
rid: Lua table with ray identification info.
surf_up: Upstream surface number. Cannot be image surface.
[surf_down]: Downstream surface number, default: next seq surface.

Get the optical path difference for a ray segment between the
first surface encounters.

All function signatures:
d = ray_opd({r = "ch"}, 3, 4) -- For sequential surfaces only
d = ray_opd({r = "ch"}, 3)    -- Same as above.
d = ray_opd({r = "ch"}, "S 3", "N 4") -- More general call.
--]]
```

- ray_pos

```
--[[
x, y, z = ray_pos(rid, surf, glb)

rid: Lua table with ray identification info.
surf: Surface identifier.
[glb]: Global reference surface, if present.
       Otherwise, local position.

All function signatures:
x, y, z = ray_pos({r = "ch"}, "S 3")    -- Local position.
x, y, z = ray_pos({r = "ch"}, "S 3", "S 1") -- Global position on
                                            -- S 3 wrt S 1.
x, y, z = ray_pos({r = "ch"}, 3, 1)     -- Same as above.
x, y, z = ray_pos({r = "ch"}, 3)        -- Local position on S 3.
--]]
```

- ray_dir

```
--[[
l, m, n = ray_dir(rid, surf, glb)

rid: Lua table with ray identification info.
```

```
surf: Surface identifier.
[glb]: Global reference surface, if present.
       Otherwise, local position.

All function signatures:
l, m, n = ray_dir({r = "ch"}, "S 3")     -- Local direction.
l, m, n = ray_dir({r = "ch"}, "S 3", "S 1") -- Global direction on
                                            -- S 3 wrt S 1.
l, m, n = ray_dir({r = "ch"}, 3, 1)      -- Same as above.
l, m, n = ray_dir({r = "ch"}, 3)         -- Local direction on S 3.
--]]
```

- list_ray

```
--[[
list_ray(rid)

rid: Table to identify ray.

List ray nodes, ray = {r = "ch", w = <wl>, z = 1, etc. }
Internally it calls ray_info function.
--]]
```

```
-- Example:
CPH> list_ray({f = 2})
SUR  X         Y         Z         L         M          GPH       T,R       A
S 2  0         0         0         0         0.1736     0         1         -
S 3  0         18.24     3.446     0         0.1736     105.1     0.9364    1
S 4  0         18.2      0.1408    0         -0.02394   107.7     0.9572    0.9992
S 5  0         15.46     0         0         -0.02836   204.1     -         1
Intensity transfer: 0.8956
Jones vector: Yamp/Xamp: 0, Yphi-Xphi: 0
```

4.8 Analysis

4.8.1 Thin Film

- stack_calc

```
--[[
res = stack_calc(params)
```

```
res: Returns a 2D numeric table of results.
params = {
    film = <filmname>,
    substrate = <sub>,
    [medium] = "AIR",
    [w] = <wl>,       -- Wavelength number, default: PWL.
    [angs] = {0.0}    -- Angles of incidence.
    [z] = 1,          -- Zoom number.
}

return table = {
 {ang, Rs, Rp, Ts, Tp, Phase_Rs, Phase_Rp, Phase_Ts, Phase_Tp}...
}
One entry for for every angle of incidence.
Phase in degrees.
--]]
-- Example script to plot stack calculations.
params = {
    film = "film1",
    substrate = "gsl1",
    medium = "AIR",
    wl  = 1,
    z = 2,
    angs = angles -- Calculated earlier, 0 to 90.
}

res = stack_calc(params)

plot_args = {
    labels = {"Rs", "Rp", "Ts", "Tp", "Rs phase", "Rp phase",
              "Ts phase", "Tp phase"},
    title = "TB",
    subtitle = "Test",
    xlabel = "Angles in degrees",
    ylabel = "R, T, Phase"
}

plot(res, plot_args)
```

Typical output of the above listing is shown in figure 4.4. The phase plots

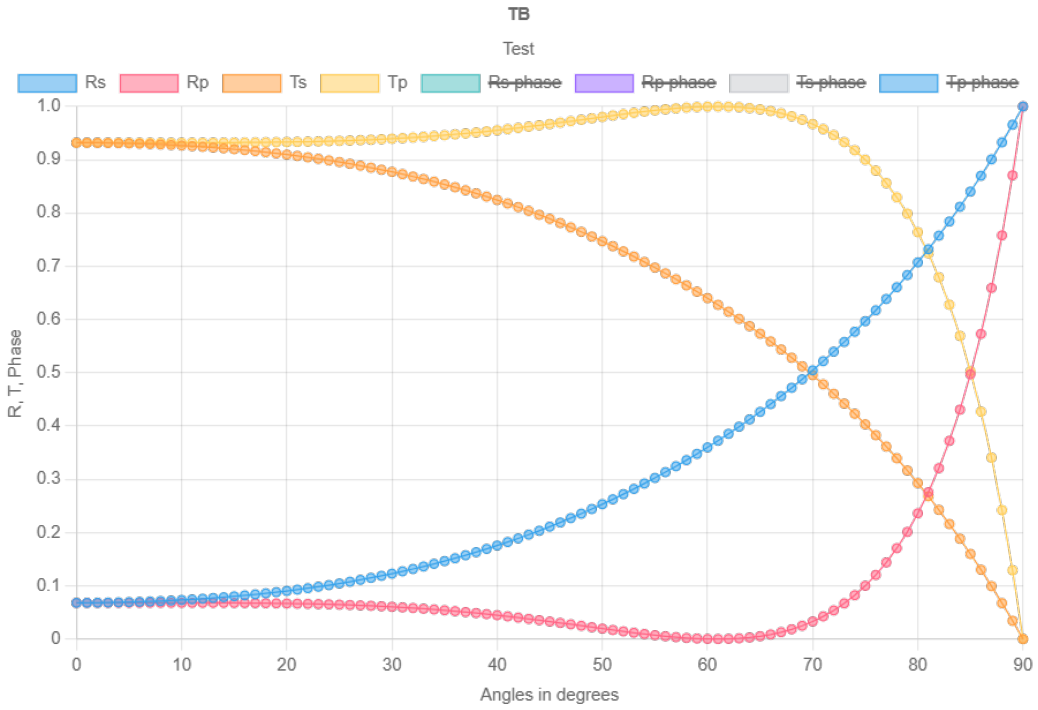

Figure 4.4: Stack calculation results.

have been unselected to make the charts more clear.

4.8.2 First Order

- `first_order`

```
--[[
res = first_order([node_tbl], num)

res: Table containing first order results.
node_tbl: Table containing ray starting information.
num: Ending surface index. 0 => surface before the image.
     It can also be zoom number, see all signatures below.

Example results table = {
     FO = {
          {x, y},    -- F1
          {x, y}     -- F2
          {x, y},    -- m_ENP, Mag to stop.
          {x, y},    -- m_XP,  Mag to XP.
          {x, y},    -- ENP_POS
          {x, y},    -- PP1
          {x, y},    -- NP1
          {x, y},    -- XP_POS
          {x, y},    -- PP2
          {x, y}     -- NP2
     }
     REF = {
            {x, y, z},           -- First ref origin.
            {l, m, n},           -- First ref Z axis.
            {x, y, z},           -- Second ref origin.
            {l, m, n}            -- Second ref Z axis.
     }
}

All function signatures:
res = first_order(nd_tbl, 5) -- Traces rays to estimate FOs.
      Ending surface is 5.
res = first_order(3) -- System FOs for zoom 3.
res = first_order()  -- System FOs for zoom 1.
```

```
--]]
```

Calculates the first order properties of the system using the ABCD matrix approach[4]. For this, base ray and rays close to the base ray are traced from the start surface (usually defined by the starting node table) to the ending sequential surface. From this ray trace, the system local ABCD matrix is estimated and first order properties follow. All first order estimations in COPHASAL follow this ABCD approach. System first order properties are estimated between the first surface after "S 1" and the surface before the image surface.

Function result depends on the signature used. When tracing rays to estimate the first order properties, the resulting table is as listed above. When system first order properties are requested, then the resulting table is like the one printed by the `list_fo` function. Note that system fist order properties are estimated only for the meridian chosen in the system settings, that is the value of the parameter `USE_XZ`.

4.8.3 Geometric

- `spot_data`

```
--[[
res = spot_data([params])

res: Return table.
params: Table of parameters.

example params = {
    [grid]   = 10,     -- Grid size, even number within [4, 30].
    [sym]    = "NONE", -- Symmetry. [NONE|ROt|YZ|xz].
    [z]      = 1,      -- Zoom number.
    [w]      = PWL,    -- Wavelength number.
    [f]      = 1,      -- Field number.
    [sur]    = "S <i>" -- Surface id. Default: image.
 }

return res = {
    rms        = 0.0,        -- RMS size.
    airy       = 0.0,        -- F1 Airy disk size for reference.
                             -- Units of mm or radians.
```

```
    rays       = 0,          -- Number of rays traced. 0 => error.
    chf2cen    = {0.0, 0.0}, -- Chief to centroid.
    uncliped   = false       -- Unclipped periphery,
                             -- True => need to check apertures.
    spots      = {  -- Spot data array.
       {x, y}...    -- Surface local tangent plane intercept
                    -- position. Scatter plot this table.
    }
 }

Generate a spot pattern and information.
--]]
-- Example:
CPH> data = spot_data()
CPH> plot(data.spots, {type="scatter"})
```

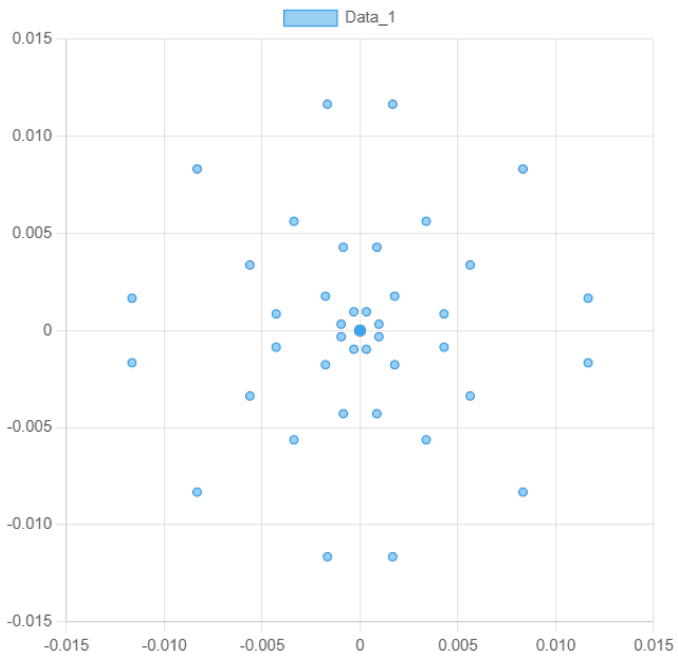

Figure 4.5: Example spot pattern.

The scatter plot generated is shown in figure 4.5.

4.9 Optimization

In the Lua interface, the optimizer is actually a new user data type called `Solver`.[5] It is initialized and used as follows.

```
--[[
opt = Solver.new([algo])

opt: Solver user type.
algo: Name of algorithm, default: "LN_COBYLA".

LN_COBYLA,   -- Constrained Optimization BY Linear Approximations.
LN_BOBYQA,   -- Bound Optimization BY Quadratic Approximations.

AUGLAG,      -- Augmented Lagrangian.

GN_DIRECT_L, -- DIRECT-L, Lipschitzian optimization.
GN_CRS2_LM,  -- Controlled Random Search with Local Mutation.
GN_ISRES,    -- Improved Stochastic Ranking Evolution Strategy.
             -- Only globall algo that supports constraints.
GN_ESCH,     -- Evolutionary Strategy with Covariance Matrix
             -- Adaptation.
G_MLSL_LDS,  -- Multi-Level Single-Linkage with Local Direct
             -- Search.

-- L stands for local algorithm and G for global.
--]]
local opt = Solver.new()
opt:set_cost(get_cost) -- Set the cost function.
opt:solve()            -- Optimize, always minimizes.
-- Note the use of ":" and "." operators.
```

Listed next are the functions of the `Solver` type. Please note that these function must be used with the ":" operator as shown above.

[5] As of version 1.7.9, the internal optimization routines are based on some of the routines from NLopt[6]. Only a subset of NLopt functionality is exposed here. Please refer to its website for more details. This feature continues to be tested.

4.9.1 Solver Setup

- `set_cost`

```
--[[
set_cost(func)

func: Name of the function with signature: double ()
      That is it returns a number and accepts no arguments.

Note: If not set, the default built in cost function (get_cost)
is used. But this only works for the default algorithm. Hence,
set the cost function.
--]]
```

- `add_eq`

```
--[[
add_eq(func, tol)

func: Function, signature double ().
tol: Tolerance.

Equality constraints always target zero, that is it will try to
make func = 0.
--]]
```

- `add_ineq`

```
--[[
add_ineq(func, tol)

func: Function, signature double ().
tol: Tolerance.

Inequality constraints always target <= zero, that is it will
try to make func <= 0.
--]]
```

- set_local_solver

```
--[[
set_local_solver(name)

name: String, name of local optimization algorithm.
      Default: LN_COBYLA

Only applicable to certain primary algorithms:
AUGLAG and G_MLSL_LDS.
--]]
```

- set_pop

```
--[[
set_pop(pop)

pop: Stochastic population size, default: 4.

Only applicable to certain algorithms:
GN_CRS2_LM, GN_ISRES, and G_MLSL_LDS.
--]]
```

- set_param

```
--[[
set_param(name, value)

name: Parameter name.
value: Parameter value.

Only applicable to certain algorithms, see NLopt documentation.
--]]
```

- set_rel_tol

```
--[[
set_rel_tol(tol)
```

```
tol: Relative tolerance for cost, default: 1e-15.

Solver will stop when the relative change in cost is less than
this value. Increase this => exit faster from optimization.
--]]
```

- set_abs_tol

```
--[[
set_abs_tol(tol)

tol: Absolute tolerance for cost, default: 1e-15.

Solver will stop when the change in cost is less than this value.
--]]
```

- set_max_eval

```
--[[
set_max_eval(max_eval)

max_eval: Maximum number of evaluations, default: 1000.

Set the maximum evaluations of the cost function.
If set to 0, the solver will only perform an evaluation run.
--]]
```

- set_time_limit

```
--[[
set_time_limit(time)

time: Maximum time in seconds, default: 60.

Set the time limit for optimization.
--]]
```

- `solve`

```
--[[
cst = solve()

cst: Retuns the final cost.

Minimizes the cost function.
--]]
```

- `set_verbose`

```
--[[
set_verbose(verb)

verb: false -> silent, otherwise verbose, default: true.
--]]
```

- `cost`

```
--[[
cst = cost()

cst: Returns cost function value.
--]]
```

4.9.2 Cost Function, Builtin

- `set_cost_params`

```
--[[
set_cost_params(params)

params =  {
 [type]    = "SPOT_SIZE", -- Cost function type.
 -- SPOT_SIZE -> variance in micron^2 or milliradian^2.

 [grid]    = 6,          -- Grid size, even number within [4, 12].
```

```
 [sym]     = "NONE",     -- Symmetry, NONE|ROt|YZ|xz.
 [wtxy]    = {1.0, 1.0}, -- Weight balance between X and Y.
 [lat_clr] = true        -- Lateral color. Set false to ignore,
                         -- useful for spectrometers.
}

Sets up the internal cost function.
--]]
```

- `get_cost`

```
--[[
cst = get_cost()

cst: Returns the internal cost function value.
     Returns a large number if something goes wrong or
     total energy in ray grid is negligible.
--]]
```

4.10 Tolerance Analysis

Function `touch` perturbs all the parameters for which there is defined a non zero tolerance.

4.11 Script Generators

Functions in this section are primarily used by the function `savefile`. However, they are documented here in case they are useful.

- `kind_of`

```
--[[
knd = kind_of(id)

knd: String representing the type of item.
id: String, item id.
```

```
Mainly useful for surfaces as there are different kinds.
Kind of the surfaces: SPHERE, NSIN, etc.
Others will simply return "DEFAULT".
--]]
```

- active_prms

```
--[[
prms = active_prms(id)

prms: Table containing active parameters of an item.
id: String id of item. Like "S 1", etc.

return = {"X", "Y", ...}
For non-denizen prms, don't specify item position, like
active_prms("W").
--]]
```

- prm_state

```
--[[
st = prm_state(id)

st: Table containing the state of the parameter.
id: String id of the parameter. Like "thi n 2", etc.

return = {
   values     = {<V1>, <V2>, ...},
   -- Values of the parameter for each zoom. Only one value is
   -- not zoomed.

   reactions  = {<R1>, <R2>, ...},
   -- Reactions of the parameter. Here R1 is another table
   -- (see below).

   variable   = {<bool>, <bool>, ...},
   -- Varying status of the parameter.

   tolerating = {<bool>, <bool>, ...},
```

```
   -- Perturbing status of the parameter.

   tolerance  = {<double>, <double>, ...}
   -- Tolerance value of the parameter.
}

R1 = {
  source_name     = "<source_name>",
  -- Name of the source parameter. Empty => not reacting.

  source_pos      = <source_pos>,
  -- Zoom number of the source.

  destination_pos = <destination_pos>,
  -- Zoom number of the destination.

  scale           = <scale>,
  -- Scale of the reaction.

  offset          = <offset>           -
  - Offset of the reaction.
}
--]]
```

- active_userglass_recipies

```
--[[

rec = active_userglass_recipies()

rec: Table of recipies for user defined glasses in use in the
     system.

return = {               -- Array of recipies.
     {
      info = {},
      -- For more information, see add_myglass docuemntation.

      data = {}
     } ...
```

```
}
--]]
```

- `all_films_z1_recipies`

```
--[[
rec =  all_films_z1_recipies()

rec: Table of recipies for all thin films defined.

return = { -- Array of recipies.
    {
      film = "filmname",
      layers = {
         {"glass_name", thickness} ...
      }
    } ...
}
--]]
```

- `global_ref_map`

```
--[[
mp = global_ref_map()

mp: Table of IDs of global references for surfaces.

Get the table connecting surface to its global reference.
return = {
     {<surf id>, <glb ref id>} ...
}
--]]
```

- `stop_surface_list`

```
--[[
stp = stop_surface_list()
```

```
stp: Table of stop surfaces for all zooms.

Get the list of stop surfaces for all zooms.

return = {<s1>, <s2> ...}
--]]
```

Chapter 5

System Settings

Aspects that affect the whole system are discussed here. These include the system parameters, wavelengths, fields, and the reference rays.

5.1 System Parameters

System parameters are listed in the table 5.1.

Parameter	Default Value	Comment
PUPIL	"EPD", ("EPD"|"ONA"|"STP")	
P_VAL	1.0	
POL_PLANE	"COLLIMATED", ("COLLIMATED"|"RAYLOCAL")	
POL_STATE	{1, 0, 0, 0}	
USE_XZ	false	
FOCAL_MODE	"FOCAL", ("FOCAL"|"AFOCAL")	

Table 5.1: System parameters and values.

5.1.1 Pupil

System pupil type is defined by the parameter `PUPIL`. It can accept the following values. `"EPD"` for entrance pupil diameter, `"ONA"` for object space numerical aperture, and `"STP"` for float by the stop maximum aperture diameter. For calculating the object space numerical aperture, the real part of

the index of refraction of "GLS2" of object surface is used. Parameter P_VAL is the value of the system pupil.

5.1.2 Polarization

Only polarized ray tracing is supported. Polarization state of the starting rays is governed by the parameter POL_STATE. It is a table of 4 numbers representing the real and imaginary part of the electric field in $\hat{x}$ and $\hat{y}$ direction. The default is pure X polarization.

When reporting and accepting ray polarization state, please note that the polarization state is in a coordinate system that is aligned with the ray such that the coordinate system $\hat{z}$ axis is aligned with the ray direction and the ray electric field vector is in the XY plane.

However such a coordinate system keeps changing with the ray direction. It is more convenient to have a coordinate system that is static with respect to the surface local coordinate system on which the ray information is being sought. Out of the two coordinate systems available, the first one ("COLLIMATED") addresses this aspect. These are associated with the parameter POL_PLANE.

Collimated

This coordinate system is fixed to the surface local system. To obtain the polarization Jones vector in this coordinate system, first the ray is hypothetically reoriented such that it is parallel to the local $\hat{z}$ axis. Then its electric field vector is decomposed along the $\hat{x}$ and $\hat{y}$ axis to calculate the Jones vector for the ray, at this surface.

Ray Local

As the name suggests, this coordinate system is local to the ray. That is the $\hat{z}$ axis is parallel to the ray direction. However, the coordinate system is rotated about the $\hat{z}$ axis such that the $\hat{y}$ axis is in a plane formed by the $\hat{z}$ axis and the surface local $\hat{y}$ axis.

5.1.3 Meridional Plane Selection

By default the first order properties of the system are estimated by constructing the ABCD matrix[4] from rays traced in the YZ meridian. However, for

some cases, like an anamorphic system or tilted system, it may be necessary to force this calculation for the XZ meridian instead. This can be done by setting the flag `USE_XZ`.

5.1.4 Focal Mode Selection

The focal mode of the system is controlled by the parameter `FOCAL_MODE`. It accepts the following values. `"FOCAL"` is used when the rays from a field are expected to converge to a spot (or diverge from one), in the image space. `"AFOCAL"` is used when the rays are expected to remain collimated in the image space.

When switching from `"FOCAL"` to `"AFOCAL"` mode, please note that spatial units in analysis functions will change to angular units and the `"SPOT_SIZE"` based internal cost function is evaluated by replacing `[x, y]` with `[l, m]`, the associated direction cosines.

5.2 Wavelengths

Parameter	Default Value	Comment
`WL`	0.5	Wavelength, micron
`WT`	1.0	Weight
`TEM`	20.0	Temperature, centigrade
`PRE`	1.0	Pressure, atm
`RH`	0.0	Relative humidity, %

Table 5.2: Wavelength related parameters and values.

Parameters associated with wavelengths are listed in table 5.2. Please note that since temperature, pressure and relative humidity directly affect the calculation of the index of refraction (along with wavelength), they are clubbed together here with wavelengths[1]. Here, `WT` is the weight of the wavelength.

Apart from these parameters, there is also the primary wavelength designation. One of the system wavelengths is designated as the primary wavelength and the first order properties are calculated for this wavelength. Func-

[1]Relative humidity is currently not included in the index calculations.

tions to set the primary wavelength is `set_pwl`. Also, wavelengths can be inserted using the function `ins_wl`.

5.3 Fields

Parameter	Default Value	Comment
X	0.0	mm or degrees
Y	0.0	mm or degrees
VXP	0.0	+X Vignetting factor
VXM	0.0	-X Vignetting factor
VYP	0.0	+Y Vignetting factor
VYM	0.0	-X Vignetting factor
WT	1.0	Weight
FTYP	"ANGLE", ("ANGLE"|"HEIGHT")	

Table 5.3: Field related parameters and values.

Parameters associated with fields are listed in table 5.3. Fields can be inserted using the function `ins_fld`. Please note that field type of "HEIGHT" is not valid when the object is far away. Maximum field angle allowed is 89.9 degrees.[2]

5.3.1 Vignetting Factors

Vignetting factors are used to adjust the entrance pupil location where the marginal rays of a field are aimed. When marginal rays are being iterated (`iterate_marginals`), these factors are ignored. Parameters VXP, VXM, VYP, and VYM define the vignetting factors. Consider figure 5.1. The marginal way that was to be aimed at the positive Y edge of the entrance pupil in the absence of any vignetting factors, is actually aimed at point (YP) in the presence of the associated vignetting factor VYP. This adjustment is described in the following equation.

$$YP = R \times (1 - VYP) \tag{5.1}$$

Here the ideal entrance pupil radius is R.

[2]Chief ray aim point related controls will be exposed with the introduction of non-spherical surfaces.

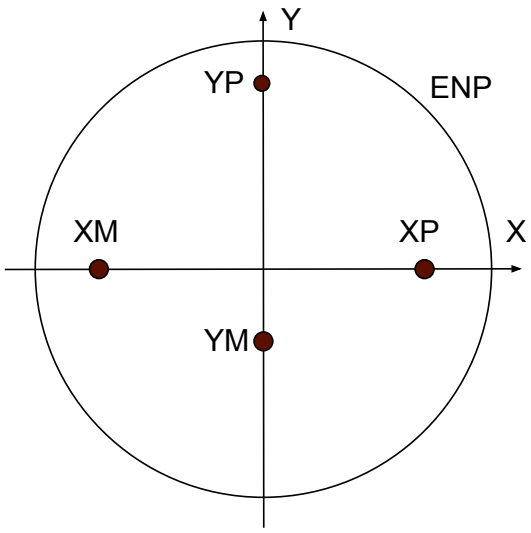

Figure 5.1: Entrance pupil (ENP), with marginal ray aim points based on vignetting factors that are positive but less than 1.0.

5.4 Reference Rays

There are 5 reference rays for every wavelength of every field defined. We will reuse figure 5.1 to describe them. Also check out the description of the function `ray_endinfo`. These rays are numbered 0 through 4 and labeled as "CH", "XP", "XM", "YP", and "YM" respectively. "CH" is the chief ray and is aimed at the center of the entrance pupil. Under ideal conditions, it will then also go through the center of the stop and exit pupil. "XP" is the marginal ray aimed at the positive X entrance pupil edge, etc.

Reference rays serve as the back bone of sequential analysis as updating them also requires updating the first order properties and assigning automatic aperture values to the sequential surfaces.

There are no parameters associated with reference rays. The functions to control these calculations are: `iterate_marginals`, `iterate_all_chiefs`, `ref_status`, and `list_fo`. By default, all chief rays are set to iterate to the center of the stop. Marginal rays are not iterated and the vignetting factors are taken into account. For most applications this is recommended.

When ever the system is changed, the reference rays are recalculated. To do so, first the chief rays are found using a multi start iterative algorithm.

The first order properties are then estimated. For this, the chief rays serve as the optical axis and the estimated pupils are perpendicular to these axis. Next, marginal rays are traced and this feeds into the next step of updating the automatic apertures of the sequential surfaces.

When the object is far away, reference rays and hence the internal analysis rays start at the next sequential surface, not the object. The assumption being that when the object is far, its surface shape and glass are not important[3]. The reference rays all start in a plane that is perpendicular to the chief ray, that is they are collimated.

[3]To model atmospheric absorption in this case, a different strategy will have to be adopted.

Chapter 6

Glasses

Any material transparent or partially transparent to optical radiation, including gases and liquids, is called glass in COPHASAL. Functions associated with manipulating and maintaining glasses are listed in 4.4. We define glasses using relative index, however, internally only the absolute index of refraction is used.

6.1 Index of Refraction

The key role of glass is to provide the index of refraction values. The index values are always kept in sync with the zoom's environmental state like temperature and pressure. Here are the supported dispersion models.

6.1.1 Interpolate

Perhaps the most common model used to define user defined glasses. Linear interpolation is used to calculate the complex index of refraction. At least one data point (wavelength index pair) is needed. Providing the imaginary part of the index is optional (it defaults to zero). No extrapolation is performed.

6.1.2 Gases

Dispersion model followed by gases including "AIR" is based on Praktische Physik[7]. This model provides only real index of refraction. The equation

is listed below (6.1).

$$n = 1 + \frac{(n_{ref} - 1) \times P}{1 + (T - T_m) \times f} \tag{6.1}$$

where:

n_{ref}	$= 1 + \left(a + \frac{b\lambda^2}{c\lambda^2 - 1} + \frac{d\lambda^2}{e\lambda^2 - 1}\right) \times 10^{-8}$
P	= system pressure
T	= system temperature
T_m	= is index measurement temperature
λ	= wavelength
a, b, c, d, e, f	= constants whose default values are valid for AIR

6.1.3 Sellmeier

The Sellmeier dispersion equation (6.2) also provides the real index of refraction. Here also λ is the wavelength.

$$n^2 = 1 + \frac{B_1\lambda^2}{\lambda^2 - C_1} + \frac{B_2\lambda^2}{\lambda^2 - C_2} + \frac{B_3\lambda^2}{\lambda^2 - C_3} \tag{6.2}$$

6.2 Absorption

There are two ways to provide material absorption. The default is to utilize the imaginary part of the index of refraction (κ) and the other is to supply a table containing the transmission measurements for a thick slab of the material. For the second approach the transmission at system wavelength is then based on linear interpolation.

6.2.1 Interpolate

When using this method, κ is ignored. This is assuming that an experimental measurement accounts not only for scattering and other losses but also for κ. When supplying the internal transmission tables, supply the slab thickness in millimeters as the first transmission entry. COPHASAL ignores the first wavelength for this purpose.

6.2.2 Ordinary

When only κ is available, the internal absorption coefficient per centimeter is calculated using equation (6.3).

$$\alpha = -\frac{4\pi\kappa \times 10000}{\lambda} \tag{6.3}$$

and

$$I = I_o e^{\alpha d} \tag{6.4}$$

where:

I = new intensity
I_o = old intensity
d = thickness, cm

6.3 Environmental Adjustments

For gases, environmental adjustment of index of refraction is already accomplished in equation (6.1).

6.3.1 Schott DNDT Equation

For optical glasses, the thermal model described in the Schott TIE document[9] is the most commonly used model. It is applicable for solids and it only accounts for the temperature changes. This approach is used to uprate the index of refraction at the specified temperature of the zoom.

6.3.2 Thermal Expansion, Omega

COPHASAL treats the coefficient of thermal expansion as a continuously varying quadratic function of temperature as shown in equation (6.5).

$$CTE(T) = C_0 + C_1(T - T_0) + C_2(T - T_0)^2 \tag{6.5}$$

where:

CTE = the coefficient of thermal expansion
T = system temperature
C_0, C_1, C_2, T_0 = parameters defining the thermal expansion

When only a single value of the CTE is available, simply assign it to C_0. The dimension change due to temperature change can then be calculated as follows.

$$\int \frac{dL}{L} = \int CTE(T)dT$$
$$let \int CTE(T)dT = \Omega(T)$$
$$=> L_2 = L_1 \times (\Omega(T_2) - \Omega(T_1))$$

We can use the function `omega` to obtain the Ω of the material associated with a surface for the zoom number.

6.4 Catalog Glasses

Catalog glasses have a special naming formate. The name is realized by joining the glass name with the catalog name with the separator being the pipe letter, `"|"`. For example, `"n-bk7|schott"` is the N-BK7 glass from Schott. User defined glasses don't have the catalog part of the name, like `"MyGlass"`.

Catalog file can be readily created once a CSV or other delimited text data file containing the glass information is obtained. To easily parse such files, there are open source Lua scripts available. Please contact us at cophasal_support@xloptix.com if you are interested in learning more.

Please note that errors may have been introduced during the generation of the glass catalog file for COPHASAL. It is highly recommended to verify the accuracy of this data against official vendor specifications before use, as discrepancies could impact results.

6.4.1 Provided Catalogs

Schott

Schott glass catalog is provided in the file "SCHOTT_glasscat.lua". This file was created from the glass data spread sheet downloaded from `www.schott.com` in 2024. Only the preferred glasses have been included.

Chapter 7

Thin Films

COPHASAL can model and optimize thin films[1]. Thin film modeling is based on the characteristic matrix method[1]. Functions associated with thin films are described in section 4.5. Function `stack_calc` is used to calculate thin film performance. COPHASAL treats thin films just like the lens system, layers can be updated on the fly, layer parameters can be addressed individually, made variable during optimization, etc. What follows is essentially a simple thin film optimization example.

7.1 Layer Parameters

Table 7.1 lists the layer parameters. In order to address the parameters, the

Parameter	Default Value	Comment
THI	0.0	
GLS2	"AIR"	

Table 7.1: Film layer parameters and values.

identifying string must also contain the film name, for example `"thi 'MgF2' L 1"`.

[1]Cements will be implemented soon.

7.2 Optimization Example

The listing below contains the script to define a single layer coating, attach it to glass "n-bk7|schott", and optimize the layer thickness to minimize reflection losses.

```
1  -- Script "filmopt.lua" to demonstrate film optimization.
2  blank()
3
4  -- User glass
5  local info = {glass = "gsl"}
6  local data = {
7      dispersion = "interpolate",
8      dispersion_data = {
9          {0.5},    -- wavelengths
10         {1.38}    -- n, MgF2 based
11     }
12 }
13 add_myglass(info, data)
14
15 -- Film
16 local stack = {
17     film = "film",
18     layers = {
19         {"gsl", 10} -- 10 nm thick, 1/4 wave ~ 90.6 nm.
20     }
21 }
22 add_film(stack)
23
24 set("gls2 s 2", "n-bk7|schott")
25 set("flm s 2", "film") -- attach film to surface 2
26
27 add_ray({pos = {0, 0}}) -- on axis user ray
28 local rid = {
29     f = 0 -- user rays, default is 1.
30 }
31
32 local res = ray_endinfo(rid)
33
34 list_tbl(res)
35
```

```
36 -- Ray power loss function
37 local function nodepowerloss()
38     -- NOTE: rays will be re-traced if any parameter is changed.
39     -- However, this is not the case with film parameters.
40     -- Hence we reset a surface parameter to set the retrace flag.
41     local thi = get("thi s 1")
42     set("thi s 1", thi)
43 
44     -- Get the ray end info
45     local n = ray_endinfo(rid) -- Retraces the ray.
46     local E = n.node[1].E
47     local nc = n.node[1].ncost
48 
49     -- Calculate power
50     local p = 0
51     for i = 1, #E do
52         p = p + E[i]^2
53     end
54     p = p * nc[1] -- Real part of ncost.
55 
56     return (1-p) -- Power loss
57 end
58 
59 print("Starting power loss: ", nodepowerloss(res))
60 
61 vary("thi film l 1")
62 
63 local opt = Solver.new()
64 opt:set_cost(nodepowerloss)
65 opt:solve()
66 
67 print("Ending power loss: ", nodepowerloss(res))
68 print("Optimal thickness: ", get("thi film l 1"))
```

Please note line 38 through 40. It is the key for such a film optimization to work. When the system is modified, a *system modified* flag is set in various objects, like the user rays container. When user ray information is sought, the object acknowledges the flag, resets it, and traces the rays afresh. However, modifying thin film parameters does not set this flag. Hence the trick used in these lines. If the function `stack_calc` is used directly, then this trickery

is not needed.

On running this script, we get the following output. The starting power loss is listed on line 19 and the final results at the end.

```
1  CPH> runfile"filmopt"
2  {
3   seq = true,
4   z = 1,
5   w = 1,
6   node = {
7    {
8     E = {0.7817, 0.1349, 0, 0, 0, 0},
9     pos = {0, 0, 0},
10    sur = "S 3",
11    phase = 2.522,
12    type = "I",
13    dir = {0, 0, 1},
14    medium = "N-BK7|SCHOTT",
15    ncost = {1.522, 0}
16   }
17  }
18 }
19 Starting power loss:    0.042349577797837
20 Optimizer               : LN_COBYLA
21 Local Algo              : LN_COBYLA
22 Dimension               : 1
23 Cost Function           : user_cost
24 Inequality Constraints  : 0
25 Equality Constraints    : 0
26 ------ Evaluation Run ------
27 Cost: 0.04235
28
29
30 ----------------------------
31 Eval No.:   43, Cost: 0.01297
32 Optimization complete, final cost: 0.012974
33 ------- Final Result -------
34 Cost: 0.01297
35
36
37 ----------------------------
```

```
Ending power loss:      0.012973661204074
Optimal thickness:      90.554884201152
```

Chapter 8

Surfaces

Surfaces are the only structural objects in COPHASAL. They define a boundary separating two (potentially different) mediums. Their properties include shape, position, aperture, diffraction effects, ability to refract, reflect, or TIR, etc. They bend the rays. Surfaces have a local coordinate system attached to them and the two medium that they separate are accessed by the parameters `GLS1` and `GLS2`. The two medium are related to the local $\hat{z}$ axis as shown in figure 8.1. `GLS2` is towards the positive $\hat{z}$ axis side of the surface. For this discussion, unless mentioned, position will imply both position as well as orientation.

8.1 Surface Lists

In COPHASAL there are two lists of surfaces and and two kinds of rays, sequential (SEQ) and non-sequential (NSEQ). Rays can change between these two modes on the fly. As the name suggests, NSEQ rays intersect with surfaces in the physical order and they can interact with all surfaces in both SEQ and NSEQ surface lists. SEQ rays on the other hand never see surfaces in the NSEQ list and they intersect with the SEQ surfaces in the sequential order in which the surfaces are listed, from the first surface (like object) to the last surface (image).

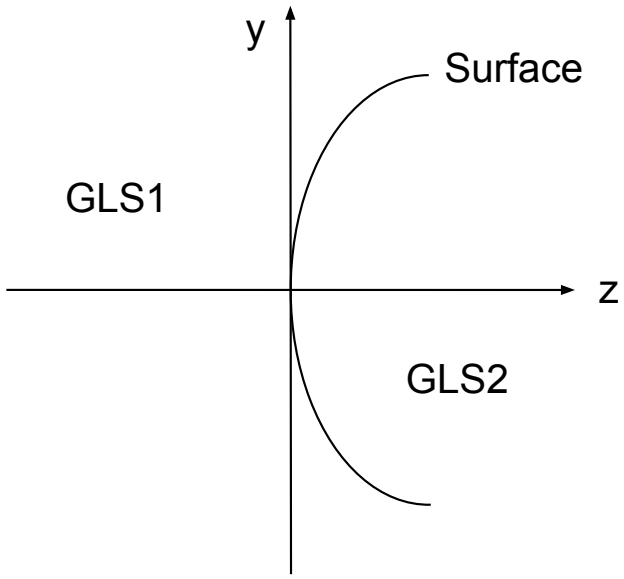

Figure 8.1: Surface YZ , its local coordinate system, and `GLS1` and `GLS2`.

8.1.1 Non-Sequential

Surfaces in this list are identified by the letter `"N"`. These surfaces can be placed in any order. They are positioned globally or with respect to a global reference. The two medium on either side of the surface must be defined with care to avoid the following ray tracing error.

Undefined_Boundary Error

Consider the figure 8.2 to understand the cause of this error. After crossing the surface boundary `"N 1"`, the ray is in medium "glass2". However, when it is incident on surface `"N 2"`, there is a mismatch between the ray medium and the incident medium on this surface. If this is the correct placement of surfaces then perhaps `GLS1` on `"N 2"` must be changed to "glass2".

8.1.2 Sequential

Surfaces in this list are identified by the letter `"S"`. SEQ rays see these surfaces in the strick sequential order. The first surface is considered the

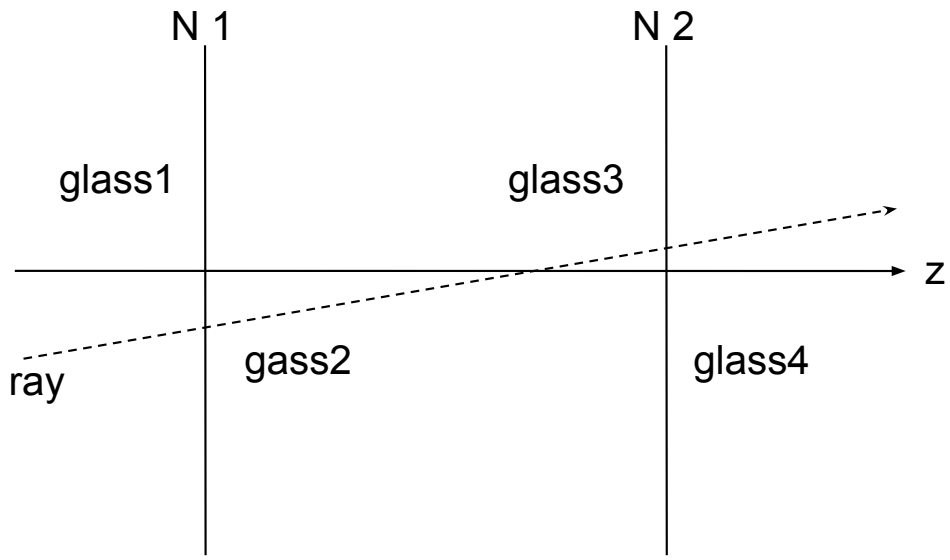

Figure 8.2: Depiction of a ray experiencing the `undefined_boundary` error on surface `"N 2"`.

object and the last one is the image. In between there has to be a stop surface. These three surfaces cannot be deleted. Hence, there are always at least three SEQ surfaces in the list. Because of the implied sequential order, there are two aspects of this surface list that make use of this fact.

Position Cascading

The thickness parameter `THI` is used by these surfaces but not by the NSEQ surfaces. Under normal circumstances, this places the next surface in the sequential order at `[0, 0, THI]`. The thickness parameter by it self will result in a train of surfaces in a line on the global $\hat{z}$ axis. If needed this can be changed by repositioning a surface resulting in a break in the position cascading and start of new train starting from the repositioned surface. A simple example depicting this is shown in figure 8.3.

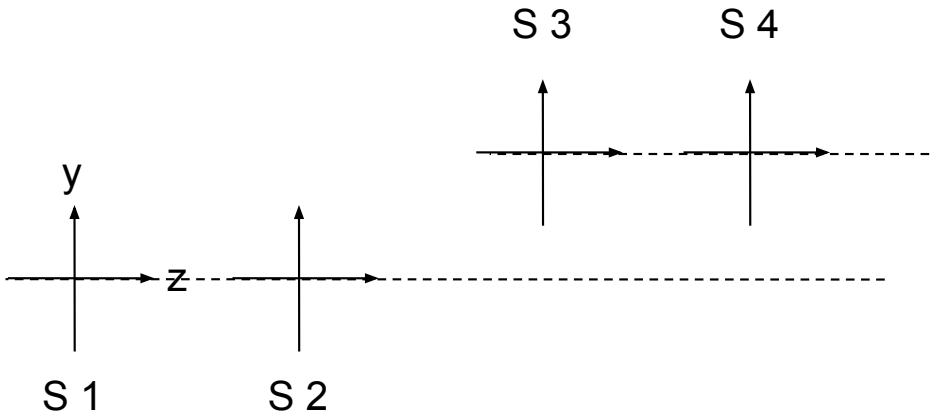

Figure 8.3: Surface `"S 3"` has been repositioned along the local $\hat{y}$ axis.

Parity Cascading

The second aspect that utilizes the sequential ordering is the internal accounting of surface parity which tracks the number of reflecting surfaces before it. The advantage of this is that for SEQ surfaces we do not have to worry about `GLS1`, it is automatically set. We only have to set the `GLS2` and for surfaces that transmit, this is the following glass and for reflecting surfaces this is the mirror substrate.

After a reflection the ray travels back in the same medium it was in before reflection. However, after a reflection the sense of the $\hat{z}$ direction reverses hence the sign of dimensional parameters like the thickness, radius of curvature, etc. must also be changed.

8.2 Common Surface Parameters

Parameters that are common to surfaces are listed in table 8.1. The following sections explain in further detail the use of these parameters.

Table 8.1: Parameters common to all surfaces.

Parameter	Default Value	Comment
X	0.0	mm, placement coordinates X
Y	0.0	mm, placement coordinates Y
Z	0.0	mm, placement coordinates Z
ALPHA	0.0	degrees, tilt about $\hat{x}$ axis
BETA	0.0	degrees, tilt about $\hat{y}$ axis
GAMMA	0.0	degrees, tilt about $\hat{z}$ axis
XOF	0.0	mm, pivot point X
YOF	0.0	mm, pivot point Y
ZOF	0.0	mm, pivot point Z
REPOS	"NONE", ("NONE"| "BASIC"| "BEND"| "RETURN"| "REVERSE")	repositioning mode
THI	0.0	mm, thickness $\epsilon[10^{-11}, 10^{11}]$
IGNR	false	rays ignore this surface
GLS1	medium	$-$z side glass name
GLS2	medium	$+$z side glass name
RFR	"TRANSMIT", ("TRANSMIT"| "REFLECT"| "TIR_ONLY")	refract mode
FLM	""	attached film name
SBS	true	film substrate is GLS1 auto sets if GLS1 is solid
TFD	{0, 0, 0, 0}	$\{x_0, y_0, c_0, c_1\}$, film taper new thickness $t = t_0 \times (1 + F)$ $F = \lvert c_0 + c_1 \times \lvert \mathbf{r} - \mathbf{r}_0 \rvert^2 \rvert$ $\mathbf{r} = [x, y]; \mathbf{r}_0 = [x_0, y_0]$
USE_JM	false	use Jones matrix, not coating
JM	{ {1, 0, 0, 0}, {0, 0, 1, 0} }	2×2 complex Jones matrix
APDZ	{0, 0, 0, 0}	$\{c_x, c_y, g_x, g_y\}$, apodization $\mathbf{e} = \mathbf{e} \times e^{\left(-g_x(x-c_x)^2 - g_y(y-c_y)^2\right)}$
DOE	"NONE", ("NONE"|	diffraction mode

Table 8.1: Parameters common to all surfaces (continued)

Parameter	Default Value	Comment
	"LINEAR")	grating
DE0	0	diffraction order
DE1..DE13		diffraction mode specific
AUT_APE		automatic aperture value
APE	"AUTO", ("AUTO"|	aperture mode
	"CIR"|	circle
	"ELL"|	ellipse
	"REC"|	rectangle
	"HEX"|)	regular hexagon
MX_APE	AUT_APE	mm, radius of the outer enclosing circular aperture. ANDs with the user defined aperture shape
DX	0.0	mm, aperture X decenter
DY	0.0	mm, aperture Y decenter
SZX	0.0	mm, aperture X size
SZY	AUT_APE	mm, aperture Y size
ROT	0.0	degree, rotation of the decentered aperture about $\hat{z}$.
OPEN	true	aperture not obscuration

8.3 Coordinate System

In the default state, "S 2" is placed at the global origin. In COPHASAL all coordinate systems and rotations are right handed. Surfaces are defined in surface local coordinate system. To place the surface at another location than the global origin, this coordinate system can be repositioned, and this is the only way NSEQ surfaces can be placed appropriately.

8.3.1 Reposition

Repositioning of a surface can be enabled by setting the REPOS parameter to the appropriate reposition mode string. By default, no repositioning is

enabled. After `REPOS` is set to any mode other than `"NONE"`, the repositioning parameters (`X`, `Y`, `Z`, `ALPHA`, `BETA`, `GAMMA`, `XOF`, `YOF`, and `ZOF`) are enabled and the surface is repositioned with respect to the prevailing coordinate system according to the mode.

`"BASIC"`

The repositioning of the coordinate system is done as follows.

1. Coordinate origin shifted to `[X, Y, Z]`
2. Coordinate system rotated by `ALPHA` about $\hat{x}$
3. Coordinate system rotated by `BETA` about $\hat{y}$
4. Coordinate system rotated by `GAMMA` about $\hat{z}$

In this case the rotations do not change the origin. However, if non zero values for the offset pivot point (`[XOF, YOF, ZOF]`) is supplied, then the tilting is performed about this offset point. This will change the coordinate origin. This is depicted in figure 8.4. In this example, `"S 2"` has been repositioned with an offset pivot point. Please note that the pivot point is given by the coordinates `[XOF, YOF, ZOF]` in the *decentered* coordinate system.

For SEQ surfaces. after this repositioning, the surface train continues as before, but along the new $\hat{z}$ axis.

`"BEND"`

This mode is same as `"BASIC"` except that for SEQ surfaces, the surface train continues along a $\hat{z}$ axis that is obtained by further *bending* the coordinate system in the same sense as defined by the tilt angles. This mode is useful when inserting fold mirrors, for example, as shown in figure 8.5. Note how the thickness parameter on `"S 2"` must be negative (because of the final $\hat{z}$ axis orientation).

`"RETURN"`

In this mode, the final coordinate system is actually the one before repositioning. That is, after repositioning the surface, we *return* back to the starting coordinate system. This is useful during tolerance analysis to perturb a single surface, for example.

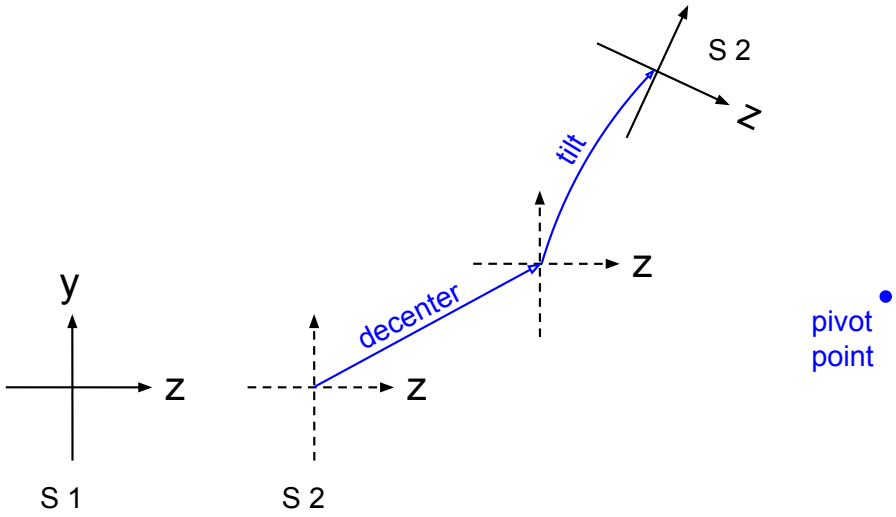

Figure 8.4: "S 2" repositioned with tilt about an offset point.

"REVERSE"

This is the *reverse* of the "BASIC" mode and can be used to undo it. In this mode, the order of operations and sign of decenter and tilt parameters is reversed. The order of operations is as follows.

1. Coordinate system rotated by -GAMMA about $\hat{z}$
2. Coordinate system rotated by -BETA about $\hat{y}$
3. Coordinate system rotated by -ALPHA about $\hat{x}$
4. Coordinate origin shifted to [-X, -Y, -Z]

Pivot point for tilting is still defined by [XOF, YOF, ZOF]. Additionally, in contrast to the "BASIC" mode, the surface local coordinate system is not affected by this repositioning, but the subsequent coordinate system is affected.

8.3.2 Global Referencing

The modes discussed in the previous section repositions the surface with respect to their nominal coordinate system. This nominal coordinate system

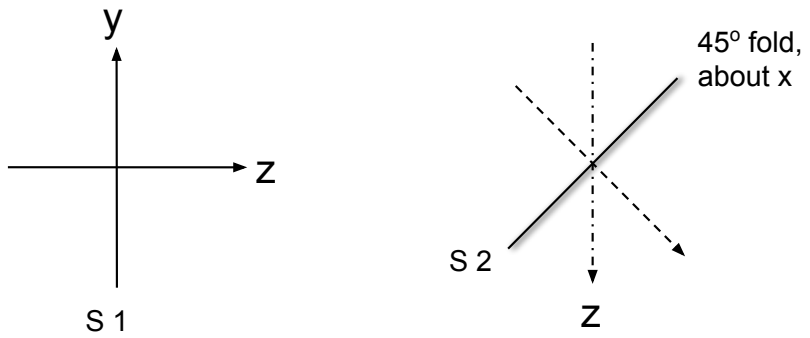

Figure 8.5: Surface 2 is mirror with reposition mode `"BEND"` and `ALPHA` of 45^o. Dashed arrow is $\hat{z}$ of the coordinate system in which mirror is defined. The centerline arrow is the final $\hat{z}$ after further bending.

can be changed to the coordinate system of another surface by globally referring to the surface. Functions to facilitate this are `set_glb_ref`, `del_glb_ref`, `list_glb_refs`, and `list_glb_position`.

If a SEQ surface globally refers to another surface, the thickness of the previous SEQ surface is then not used. A common use of this functionality is to isolate groups of element. There are some limitations on setting up global referencing.

- A surface can refer only backwards in the list. Example, `"N 4"` can refer to `"N 2"`, but not the other way round.
- SEQ surface cannot refer to NSEQ surfaces. Example, `"N 3"` can refer to `"S 5"` but the other way round.

8.3.3 Best Practice

Models with complex repositioning schemes can be hard to understand and bad for posterity. The later can be far more important. For such reasons, please keep repositioning schemes simple. If possible, maintain symmetry about a meridian. As a convention, we will use the YZ meridian. Here are few specifics to note.

- For sequential surfaces, avoid large tilts that can flip the direction of $\hat{z}$, which can result in the undefined boundary error.
- COPHASAL accounts for the relative `GAMMA` rotation between the start and ending surface when estimating ABCD matrix, however, large relative rotation can lead to first order calculation errors. For this reason, it is better to isolate `GAMMA` rotations. However, `AUT_APE` estimates are still accurate.

8.4 Optical Properties

8.4.1 Refract Mode

The refract mode of a surface can be changed by setting the `RFR` mode. Note that for SEQ surfaces, after a reflection the signs of quantities like thickness

change. For reflecting surfaces, GLS2 acts as the substrate[1].

8.4.2 Diffractive Properties

A surface can be converted into a diffractive optical element by changing the DOE mode to something other than "NONE"[2]. After this change, surface parameters DE0 and higher are enabled.

DE0 is the diffraction order.

"LINEAR"

Use this mode to model linear gratings. This mode also needs parameters DE1 through DE4.

DE1 is the grating pitch in mm, default is 0.0 which means that the default grating frequency is also 0.0.

[DE2, DE3, DE4] defines a vector parallel to the grating vector. Its default value is [0, 1, 0], that is the vector is pointing in the surface local $\hat{y}$ direction. Grating vector is internally normalized.

The pitch and the grating vector defines three dimensional grating planes. The intersection of these planes with the surface results in the *fringe* or *rulings* on the surface. The local fringe pattern at the ray surface intersection point is used to diffract the ray.

Please note that diffraction efficiency is not calculated. It is assumed that the efficiency is 100%.

8.4.3 Thin Film and Related Properties

Thin Film

Films are attached to the surface by assigning the film name to the FLM parameter. The film must first be defined before attaching to surface, see function add_film.

[1] Currently, COPHASAL does not automatically assign a metal to GLS2. It must be assigned a glass manually.

[2] Currently only one other mode is supported that is the linear grating.

If `GLS1` is solid, COPHASAL automatically assumes that this is the substrate for the film. However, this can be change by setting the `SBS` flag. Set it to true of film substrate is `GLS1`. This can be important when considering reflections and there are absorbing layers in the film stack.

Tapered Film Definition

Thin film manufacturing process can result in layer thickness variation that is a function of the lateral position on the surface. This can modeled with the `TFD` parameter which accepts a table of 4 numbers, $\{x_0, y_0, c_0, c_1\}$. The relationship between the nominal thickness t_0 and the actual thickness used for calculation is given by equation (8.1).

$$t = t_0 \times \left(1 + |c_0 + c_1 \times |\mathbf{r} - \mathbf{r}_0|^2|\right) \tag{8.1}$$

Here $\mathbf{r} = [x, y]$, the lateral position on the surface, and $\mathbf{r}_0 = [x_0, y_0]$. Equation (8.1) represents a parabolic thickness error centered about $\mathbf{r}_0$. Note that if $\mathbf{r}_0$ is placed far away, the taper approximately becomes a wedge.

Jones Matrix

It is possible to apply a Jones matrix to the complex electric field of the ray interacting with the surface, however, it cannot be applied when there is a film attached to the surface. To apply a Jones matrix, first set the `USE_JM` flag to true and then supply the Jones matrix by updating the parameter `JM` with a table of tables. Here is an example of updating this parameter.

```
set("use_jm s 2", true)
JM = { {jm11, jm12, jm13, jm14},
       {jm21, jm22, jm23, jm24} }
set("jm s 2", jm)
```

Here is how this is interpreted.

$$\mathbf{JM} = \begin{bmatrix} jm_{11} + jm_{12}\mathbf{i} & jm_{13} + jm_{14}\mathbf{i} \\ jm_{21} + jm_{22}\mathbf{i} & jm_{23} + jm_{24}\mathbf{i} \end{bmatrix}$$

The electric field is updated as follows.

$$\mathbf{e}^{(new)} = \mathbf{JM} \times \mathbf{e} \tag{8.2}$$

$$where$$

$$\mathbf{e} = \begin{bmatrix} \tilde{e_x} \\ \tilde{e_y} \end{bmatrix}$$

In equation (8.2), **e** is the complex electric field vector with no component in the $\hat{z}$ direction. This is because this calculation is performed in a polarization coordinate system in which the ray direction is parallel to the $\hat{z}$ axis. This coordinate system is defined by setting the parameter `POL_PLANE` (see table 5.1).

8.4.4 Apodization

Surfaces can be set to impart apodization to the incident electric field. To enable apodization set the parameter `APDZ` with non zero values of g_x and g_y. The electric field amplitude change is described in equation (8.3). Here, $[c_x, c_y]$ is the center of the Gaussian apodization.

$$\mathbf{e}^{(new)} = \mathbf{e} \times exp\left(-g_x(x - c_x)^2 - g_y(y - c_y)^2\right) \tag{8.3}$$

8.5 Aperture

Apertures define the shape of the physical extant of the surface through with the rays can pass. It is actually the projection of the opening, along $\hat{z}$, on the XY plane. SEQ rays that are outside of the aperture are not blocked but are sapped of all energy. NSEQ rays that are out of aperture, miss the surface. Apertures are defined in the surface local coordinate system. Broadly, there are two kinds of apertures, automatic aperture and user defined aperture.

8.5.1 Automatic Aperture

The automatic aperture has a circular shape, is centered, and is setup to be the parameter `AUT_APE` of the surface. It is not user modifiable. It is automatically set by the system reference rays to pass all marginal rays. For the stop surface, if marginal rays fail to increase this radius from default of 0.0, this value is set to be the ideal stop size based on system pupil settings.

8.5.2 User Defined Apertures

To define a user defined aperture, the `APE` mode must be changed from `AUTO` to the desired mode.

All user defined modes are composed of two shapes, an outer centered circular shape with radius `MX_APE`, and an inner shape defined by the mode

and parameters `DX`, `DY`, `SZX`, `SZY`, `ROT`, and `OPEN`. The actual aperture is the logical AND between these two shapes.

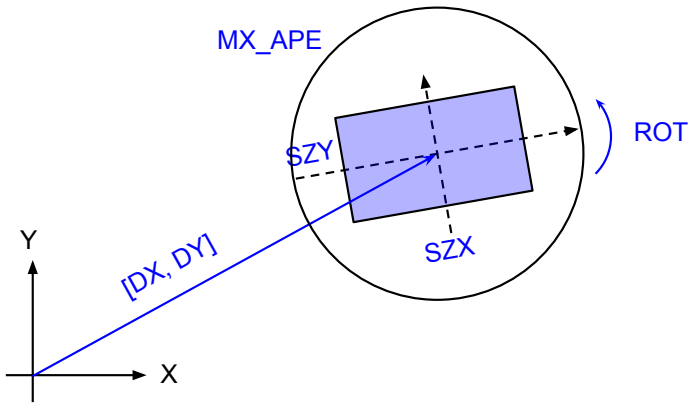

Figure 8.6: Example of setting up a rectangular aperture. The shaded rectangle is the actual aperture realized as outer circle AND rectangle.

The outer shape is always open inside and blocks outside. The inner shape is by default open inside but this can be changed to being open outside and blocking inside by setting `OPEN` to `false`. An example of a rectangular aperture is shown in figure 8.6. The following are the supported aperture modes.

`"CIR"`

This is a circular aperture. Parameters `SZX` and `SZY` have special meaning, while `ROT` has no effect for this mode. This mode can also describe an annular aperture as shown in figure 8.7.

`SZX` is the inner radius of the annulus. Default value of 0.0 means that it is a simple circular aperture without a central obscuration. It is intended not to exceed `SZY`.

`SZY` is the outer radius of the annulus. Default is `AUT_APE`.

When changing from `"AUTO"` to `"CIR"`, the automatic aperture is converted to a circular aperture.

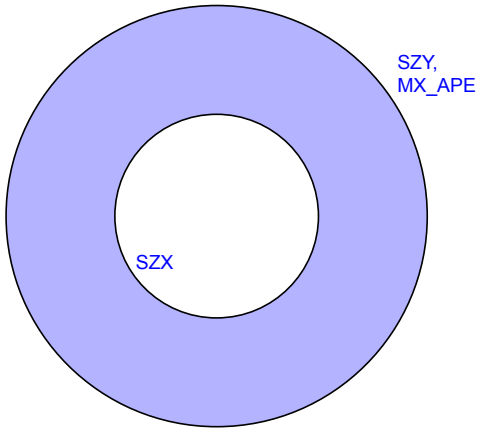

Figure 8.7: Example of an annular aperture. `SZY` is the outer radius and `MX_APE = SZY`. Inner radius is `SZX`.

`"ELL"`

This mode describes an elliptical aperture. Its setup is similar to that of `"REC"`, see figure 8.6.

`"REC"`

This mode describes a rectangular aperture. See figure 8.6. Parameters `SZX` and `SZY` are the semi widths along the respective axis.

`"HEX"`

This mode describes a hexagonal aperture. This mode does not make use of the `SZX` parameter. An example of this aperture shape without any rotation is shown in figure 8.8.

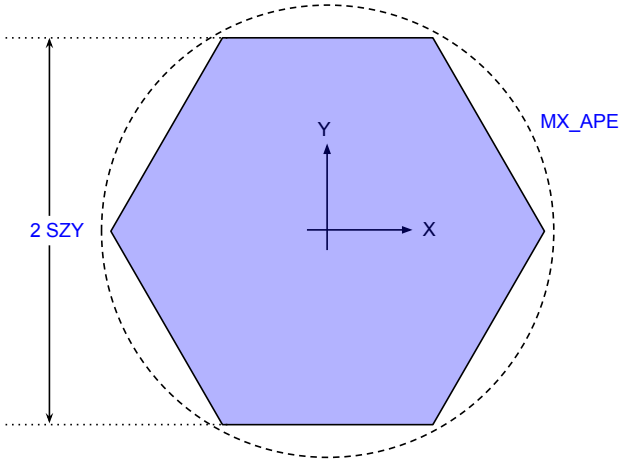

Figure 8.8: Hexagonal aperture with out any `ROT`.

8.6 Ignore Surface

In COPHASAL, all rays will ignore a surface whose `IGNR` flag has been set to `true`. Note that only rays will stop seeing the surface, however, the surface will continue to make its presence felt to other surfaces. For example, `THI` of an ignored SEQ surface still matters. Object, stop surface, and image cannot be ignored. An example of the use of this feature is to model optical elements that can be *flipped* in and out of the optical train.

An important consequence of ignoring a surface is the potential of introducing the undefined boundary error (8.1.1). For this reason, it is recommended to ignore entire optical elements, like both surfaces of a lens.

Chapter 9

Surface Types

In addition to the common surface parameters (table 8.1), each surface type can support additional parameters and associated functionality. Different surface types identified by the `type` table entry can be inserted using the function `ins_surf`. This chapter lists the different surface types available in COPHASAL.

9.1 Sphere

A standard spherical shape is provided by the surface type `"SPHERE"`. The additional parameters supported by spheres are listed in table 9.1.

Parameter	Default Value	Comment
`CUY`	`0.0`	1/mm, curvature
`RDY`	`0.0`	mm, radius of curvature. Not a real internal parameter. Only for help with setting the curvature. `RDY = 0.0 => CUY = 0.0`.

Table 9.1: Additional parameters of the spherical surface.

9.2 Conic

A conic surface shape is provided by the surface type `"CONIC"`. The additional parameters supported by conics in addition to those supported by spheres

Parameter	Default Value	Comment
CONY	0.0	conic constant, k

Table 9.2: Additional parameters of the conic surface.

are listed in table 9.2. The sag of the conic surface is given by equation (9.1).

$$z = \frac{cr^2}{1 + \sqrt{1 - (1+k)c^2r^2}} \tag{9.1}$$

where:

c = curvature
$r = \sqrt{x^2 + y^2}$
k = conic constant

$k < -1$	$k = -1$	$-1 < k < 0$	$k = 0$	$k > 0$
Hyperboloid	Paraboloid	Ellipsoid	Sphere	Oblate Ellipsoid

Table 9.3: k values and conic surface shape.

9.3 QCON Asphere

Parameter	Default Value	Comment
NRAD	0.0	mm, normalization radius, Default is AUT_APE or 1.0
EPm	0.0	mm, coefficient for the asphere polynomial term of order $2m + 4$. with $m <= M$. For example, parameter for the coefficient of order 4 polynomial is EP0 as $m = 0$

Table 9.4: Additional parameters of the qcon surface.

The QCON asphere surface shape[3, 2] is defined by surface type "QCON". The additional parameters supported by QCON asphere in addition to those

supported by conics are listed in table 9.4. The sag of the QCON surface is given by equation (9.2).

$$z = \frac{cr^2}{1 + \sqrt{1 - (1+k)c^2r^2}} + u^4 \sum_{m=0}^{M} EPm \times Q_m^{con}(u^2) \tag{9.2}$$

where:

u	$= r/nrad$
$nrad$	= normalization radius
$Q_m^{con}(x)$	= polynomial of order m
EPm	= extra parameter `EPm`
M	= max order of polynomial

The maximum order of the polynomials, M, is 6. Figure 9.1 shows the first few terms. Only the absolute value of the normalization radius is considered.

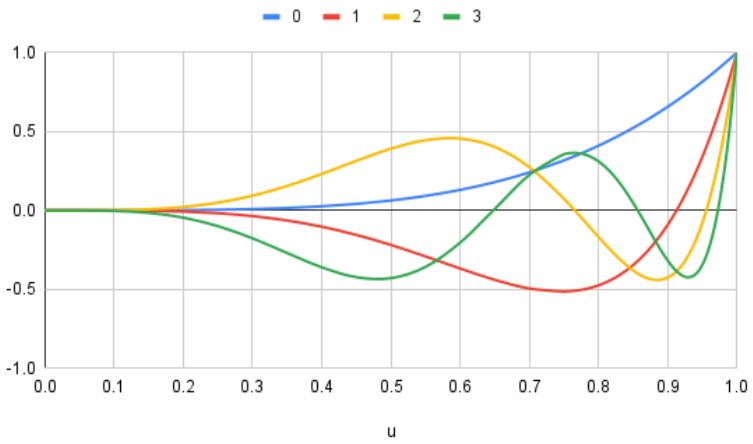

Figure 9.1: $u^4 Q_m^{con}(u^2)$ as a function of u for various values of m.

9.4 Toric

This is the toroidal surface shape, defined by the surface type `"TORIC"`. The YZ profile of a toric surface is the same as that of the QCON asphere and

Parameter	Default Value	Comment
CUX	0.0	1/mm, curvature
RDX	0.0	mm, radius of curvature. Not a real internal parameter. Only for help with setting the curvature. RDX = 0.0 => CUX = 0.0.

Table 9.5: Additional parameters of the toric surface.

is obtained by replacing the radial position r with the local y ordinate in equation (9.2). The surface is realized by revolving this YZ curve about an axis parallel to the $\hat{y}$ and located at [0, 0, RDX]. With RDX of 0.0 and no other aspheric contributions, we obtain a cylinder whose axis is $\hat{x}$.

Please note that this surface is ill-defined when the YZ profile of the surface intersects the axis of revolution. No attempt is made to check for this situation.

9.5 NSIN and NSOUT

These surfaces are primarily intended to toggle the sequential mode of a ray intersecting with them. The ray must intersect within the aperture. "NSIN" converts a ray to a NSEQ ray and "NSOUT" converts the ray to a SEQ ray. These surfaces can only be placed in the sequential surfaces list. A typical use case of these surfaces is to enclose a NSEQ group of surfaces starting from a "NSIN" and ending with a downstream "NSOUT" surface. However, once a ray is converted to NSEQ mode, it can potentially even skip the "NSOUT" surface.

It is important to consider the difference in behavior of NSEQ rays compared to SEQ rays when it comes to apertures. When SEQ rays fall out of aperture, their energy is sapped but they continue on tracing towards the image surface. NSEQ rays on the other hand, skip the surface and there is no entry of their intersection on that surface[1]. Hence, the algorithm to set the automatic apertures on such a surface will skip the surface as well. It is thus advisable to set user defined apertures for such cases.

[1]It is possible that a NSEQ ray has a chance to interact with a surface multiple times, but for the purposes of reference rays, only the first interaction matters.

Chapter 10

Analysis

Most analysis features involve tracing rays, either user-defined or internal rays. Apertures, whether automatic or not, are always taken into account. For functions that trace a grid of rays across the pupil, the ray grid overfills the pupil in the expectation that some aperture in the system will block the rays in periphery of this grid. If any of these peripheral rays is not blocked by an aperture, the function sets the flag *unclipped* to true in the return value. This is usually a sign that the apertures are not in step with the pupil value.

10.1 Ray Tracing and Analysis

Functions associated with ray tracing and information access are described in Section 4.7.

10.1.1 Starting of Rays

Rays are defined in the local coordinate system of the surface. The input of the electric field is interpreted according to the parameter `POL_PLANE` and the starting power is unity.

Here are the steps taken before the starting node of a ray is calculated.

1. The actual ray surface intersection is calculated.
2. In the presence of coating, the ray intersection point is adjusted. See figure 1.1.

3. The electric field is normalized by the real part of the index of the medium. That is $\mathbf{e}^{new} = \mathbf{e}/\sqrt{n_r}$.

4. The ray is propagated to the adjusted intersection point, updating the electric field and geometric phase in the process.

The ray starting medium is assumed to be homogeneous and isotropic. An upper limit of 1000 is imposed on the number of segments in a ray. The upper limit on the angle of incidence is 89.9 degrees.

10.2 First Order Analysis

Function `first_order` provides the first order cardinal points of the system between two surfaces. Also see function `list_fo`.

10.3 Geometric Analysis

10.3.1 Spot Size

Function `spot_data` generates spot data and information. Only rays with non zero energy are considered and the spot variance calculations are based on energy weighted moments. For example, the first moment for x is given by the equation (10.1). Here, ϵ_i is the energy in the individual ray.

$$< x >= \frac{\sum (\epsilon_i x_i)}{\sum \epsilon_i} \tag{10.1}$$

10.4 Thin Film Analysis

Function `stack_calc` calculates the s and p amplitude reflectance and transmittance.

Chapter 11

Optimization

Optimization in COPHASAL[1] is performed by first instantiating and updating a user data type called the `Solver`. Then invoking the function `solve` of the `Solver` data type. Functions associated with optimization are listed in section 4.9.

Every optimization problem needs a cost function to minimize and a set of variable parameters to manipulate, in order to achieve the minimization goals. In COPHASAL, variables are defined by designating parameters as variables using the function `vary`. Without the cost function and variables, minimization cannot proceed.

11.1 Constrained Optimization Example

An example of using the `Solver` for general function optimization is shown in the listings below. The cost function is given by the equation $(x-1)^2+(y-2)^2$. This has to be minimized such that $y \geq 4 - x$. This problem is described in the figure 11.1.

```
-- Contents of file test.lua
blank()

-- For variables, we will use the DE parameters of a surface
-- with DE0 = 0. Hence the optical system will not be affected.
```

[1]This feature is currently based on NLopt[6] and is still being tested and updated. Only LN_COBYLA, and to a lesser extant LN_BOBYQA, have been tested so far. Please see NLopt documentation for details.

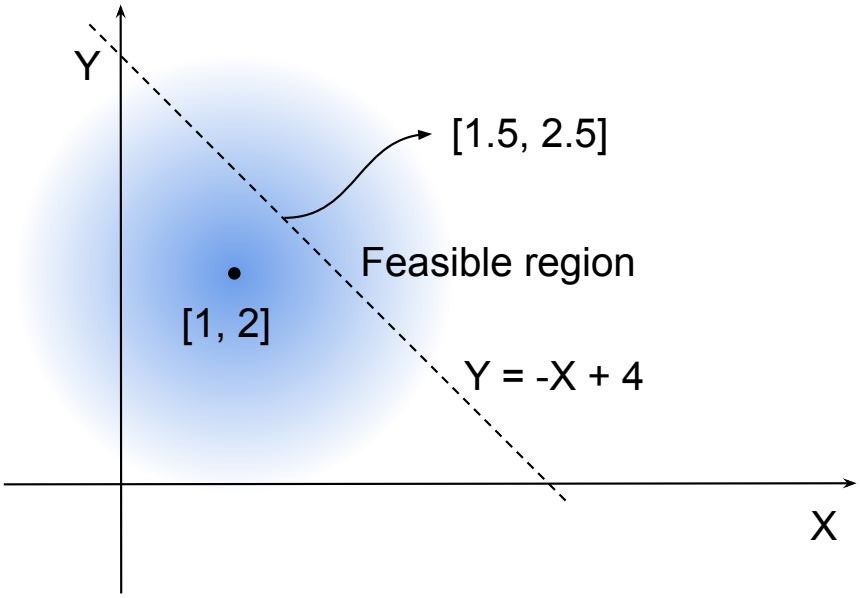

Figure 11.1: Graphical depiction of the optimization example.

```
set("doe s 2", "linear")
vary("de1 s 2") -- x
vary("de2 s 2") -- y

local center = {1, 2} -- Center point.

-- The cost function to minimize.
-- Minimum of 0 is at the center point.
local function my_cost()
  local x = get("de1 s 2")
  local y = get("de2 s 2")

  local r = {x - center[1], y - center[2]}

  return r[1]*r[1] + r[2]*r[2]
end

-- First unconstrained optimization.
local o = Solver.new("LN_COBYLA") -- Default.
```

```
o:set_cost(my_cost)
o:set_verbose(false)
o:solve()

-- Results.
print("Unconstrained optimization:")
print("Final cost: " .. o:cost())
print("At [x, y] = ["..get("de1 s 2")..","..get("de2 s 2")..."]")

-- Now we will add an inequality constraint.
-- y >= -x + 4 => y + x - 4 >= 0
local function ineq()
    local x = get("de1 s 2")
    local y = get("de2 s 2")

    return 4 - y - x
    -- Solver always tries to set ineq() <= 0
    -- Hence we need to negate the inequality.
end
o:add_ineq(ineq, 1e-6)

-- Constraint optimization.
o:solve()

-- Results.
print()
print("Constrained optimization:")
print("Final cost: " .. o:cost())
print("At [x, y] = ["..get("de1 s 2")..","..get("de2 s 2")..."]")
```

```
CPH> runfile"test"
Unconstrained optimization:
Final cost: 3.7185048536536e-16
At [x, y] = [0.99999999921645,2.0000000192675]

Constrained optimization:
Final cost: 0.5
At [x, y] = [1.5000000097274,2.4999999902726]
```

11.2 Solver Data Type

Instantiating and setting the `Solver` type is described in section 4.9. Here some of these aspects will be discussed in further detail.

11.2.1 Solver Algorithms

The "L" in the algorithm names stand for local optimizers and "G" stands for global optimizers. The following are the algorithms that have gone through basic testing.

`LN_COBYLA`

This algorithm supports constraints.

`LN_BOBYQA`

This algorithm does not directly support constraints.

`AUGLAG`

This is the Augmented Lagrangian algorithm and its main use is to enable constraint management for other optimization algorithms. For example, a `Solver` can be created with this as the primary algorithm and `LN_BOBYQA` can be designated as the local optimizer (see function `set_local_solver`). This way, we will have constrained optimization even with the `LN_BOBYQA` algorithm[2].

11.2.2 Settings

The maximum number of cost function evaluations can be set using the function `set_max_eval`. Set it to 0 to perform an evaluation run. The system will not be changed in such an optimization run. But the cost function and the constraints functions will be evaluated. Any inconsistencies and errors during this evaluation run should be addressed before performing the actual optimization.

[2]Currently, the convergence with this algorithm is slow.

11.3 Cost Functions, Builtin

The internal cost function value is obtained from `get_cost`. The internal cost function is setup using `set_cost_params`.

`"SPOT_SIZE"`, is the default cost type and provides the weighted mean spot variance. Its calculation is described in the following equations. The variances are calculated using ray energy weighted moments. Let the spot variance in $\hat{x}$ for a particular wavelength, field, and zoom number be given by $\sigma_x^2(w, f, z)$. When lateral color is being considered, the primary wavelength centroid for the field and zoom is chosen as the reference origin in moments calculations. Otherwise it is the spot centroid for the current wavelength itself. When lateral color is ignored, each spot size is minimized independently, useful in a spectrometer for example.

The combined spot variance is calculated as shown below.

$$\sigma^2(w, f, z) = WT_{xy}[1]\sigma_x^2(w, f, z) + WT_{xy}[2]\sigma_y^2(w, f, z)$$

Here WT_{xy} is the `wtxy` table entry. This can be used to shift the balance between the X and Y aberrations components, useful for anamorphic systems for example.

For calculating the total cost function, let $WT_w(z)$ be the weight of a wavelength at a zoom number and $WT_f(z)$ be the field weight. The total variance is calculated by equation (11.1).

$$\sigma^2 = \frac{\sum_{w,f,z} \left(\sigma^2(w, f, z) \times WT_w^2(z) \times WT_f^2(z)\right)}{\sum_{w,f,z} \left(WT_w^2(z) \times WT_f^2(z)\right)} \tag{11.1}$$

Chapter 12

Tolerancing

Every practical design must account for manufacturing limits and natural fabrication errors by performing a tolerance analysis. Skip this step and there will be reckoning, hopefully before the manufacturing begins.

Functions that support tolerance analysis are listed in section 4.1.3. There are two modes of performing tolerance analysis, linear model based[1], and script driven Monte Carlo simulations.

12.1 Monte Carlo Simulations

Monte Carlo simulations in COPHASAL are driven by user scripts and allow full control. A simple example follows.

```
--[[
First, define a singlet lens, one wavelength and field.
--]]
blank()

-- System settings
set("P_VAL", 20, 1)

-- Sequential surfaces
set("THI S 1", 1e11);

set("THI S 2", 100.0);
```

[1]The linear model based tolerance analysis coming soon.

```
ins_surf("S 3", {type = "SPHERE",
                 glass = "N-BK7|SCHOTT",
                 rdy = 50.0,
                 thi = 5.0} )

ins_surf("S 4", {type = "SPHERE",
                 glass = "AIR",
                 rdy = 1176.0,
                 thi = 96.57} )

set_stop(2)

-- Variables
vary("cuy s 3")
vary("cuy s 4")

-- Optimize
local opt = Solver.new()
opt:set_cost(get_cost)
opt:solve()

-- Starting results.
local data = spot_data()
-- plot(data.spots, {type="scatter"})

local ideal_rms = data.rms
print(string.format("Ideal RMS: %.3f um", ideal_rms*1000))

--[[
Perform basic tolerance analysis.
Define tolerances, run a Monte Carlo simulation
and plot the results.
--]]
-- Define tolerances
set_tol("thi s 4", 0.25)
set_tol("cuy s 3", 1e-4)
set_tol("cuy s 4", 1e-4)

-- Monte Carlo Simulation
```

```
local trials = 1000
local rms = {}
for i = 1, trials do
  touch()
  local data = spot_data()
  rms[i] = data.rms
  undo()
end

-- Plot results
local histdata = hist_table(rms)
plot(histdata, {type="bar",
                title="RMS Histogram",
                xlabel="RMS (mm)",
                ylabel="Frequency",
                labels = {"RMS Spot Size"} } )
```

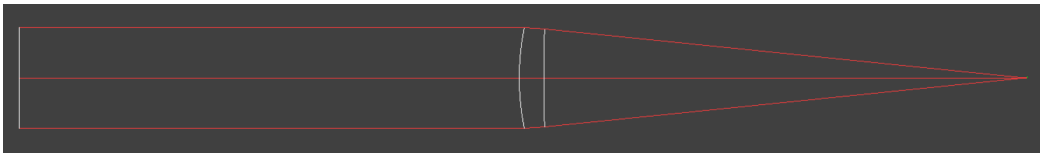

Figure 12.1: Layout of the lens ready for Monte Carlo simulations.

The optimized lens layout is shown in figure 12.1. The surface listing is shown next.

```
CPH> list"s"
S:
 Z1:     LBL        TYP       RDY          THI          MED          RFR      AUT_APE...
 S 1             SPHERE         0        1e+11          AIR     TRANSMIT            0
 S 2             SPHERE         0          100          AIR     TRANSMIT           10
 S 3             SPHERE     50.02  v         5    N-BK7|SCH     TRANSMIT           10
 S 4             SPHERE      1004  v     96.57          AIR     TRANSMIT        9.719
 S 5             SPHERE         0            0          AIR     TRANSMIT      0.03897
Stop: 2
```

Listing 12.1: Listing of sequential surfaces.

The script output follows and the resulting histogram plot is shown in figure 12.2.

```
CPH> runfile"test"
Optimizer                : LN_COBYLA
Local Algo               : LN_COBYLA
```

```
Dimension              : 2
Cost Function          : user_cost
Inequality Constraints : 0
Equality Constraints   : 0
------ Evaluation Run ------
Cost: 3193



-----------------------------
Eval No.: 1000, Cost: 338.7
Optimization complete, final cost: 338.710535
------- Final Result -------
Cost: 338.7



-----------------------------
Ideal RMS: 19.299 um
```

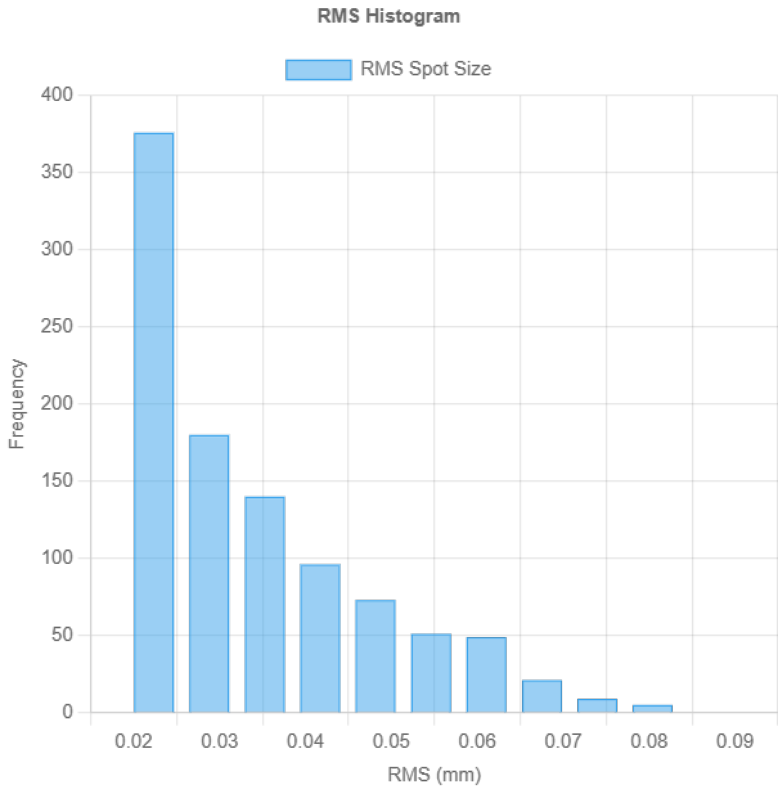

Figure 12.2: Histogram of the Monte Carlo simulation.

Appendix A

Error Codes

A.1 Ray Trace Errors

Error Code	Meaning
thick_metal_layer	Thick metal layer in film
thick_TIR_layer	Thick TIR layer in film
null_index	Null index of refraction
unexpected_TIR	No transmit, unexpected TIR
could_not_TIR	No TIR in a TIR only mode
could_not_diffract	No diffraction in order
negligible_power	Ray power too low
too_many_segments	Too many segments in ray
skips_surface	Ray does not see the surface
skips_all_surfaces	Ray misses all surfaces
too_steep_aoi	Angle of incidence too steep
undefined_boundary	Ray and surface medium mismatch
illegal_surface	Blanket code for illegal input
iterator_trouble	Blanket code for iteration failures
arithmetic_fault	A GRIN can cause this

Table A.1: Ray trace error codes that the user can expect to encounter.

A.2 Index Calculation Errors

Error Code	Meaning
WL_OUT_OF_RANGE	Wavelength out of range
TEM_OUT_OF_RANGE	Temperature out of range
PRE_OUT_OF_RANGE	Pressure out of range
RH_OUT_OF_RANGE	Relative humidity out of range
SCHOTT_ARITH_FAULT	Arithmetic fault, Schott DNDT
SELLMEIER_ARITH_FAULT	Arithmetic fault, Sellmeier

Table A.2: Index of refraction calculation error codes that the user can expect to encounter.

Appendix B

Software Libraries Used

COPHASAL has leveraged external software libraries. A list of these libraries and their associated license texts can be found in the accompanying document, software_libraries.pdf.

Index

References

[1] M. Born and E. Wolf. *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*. Cambridge University Press, 1999.

[2] G. W. Forbes. "Robust and fast computation for the polynomials of optics". In: *Opt. Express* 18.13 (June 2010), pp. 13851–13862. DOI: 10.1364/OE.18.013851. URL: https://opg.optica.org/oe/abstract.cfm?URI=oe-18-13-13851.

[3] G. W. Forbes. "Shape specification for axially symmetric optical surfaces". In: *Opt. Express* 15.8 (Apr. 2007), pp. 5218–5226. DOI: 10.1364/OE.15.005218. URL: https://opg.optica.org/oe/abstract.cfm?URI=oe-15-8-5218.

[4] A. Gerrard and J.M. Burch. *Introduction to Matrix Methods in Optics*. Dover Books on Physics. Dover, 1994.

[5] R. Ierusalimschy. *Programming in Lua*. 4th ed. Lua.org, 2016.

[6] Steven G. Johnson. *The NLopt nonlinear-optimization package*. https://github.com/stevengj/nlopt. 2007.

[7] F. Kohlrausch. *Praktische Physik vol 1*. B.G. Teubner, 1968.

[8] Lua. https://www.lua.org/.

[9] The SCHOTT Group. *TIE-19 Temperature Coefficient of the Refractive Index*.