LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type **LIGO-T970193-00 -** **E** Nov. 1997

Overview of End-to-End model

Biplab Bhawal, Matt Evans, Edward Maros,
Malik Rahman and Hiro Yamamoto

Distribution of this draft:

xyz

This is an internal working note
of the LIGO Project..

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

LIGO-DRAFT

1 ABSTRACT

This document presents an overview of the End to End model: (1) physics and motivation of this model, (2) the underlying program structure, (3) how to use it and (4) how to expand it. This document also includes the road map guiding you to other references which may provide you more detailed information. Tutorials given in this document will cover most of the essential information, and one can start using the End to End model with this document as a quick reference.

2 KEYWORDS

End to End model, Adlib, Alfi, Optics, Summation Cavity, $A_{3,10,10}$, E2E, C++

3 OVERVIEW

3.1. Background

In the past, simulation softwares for LIGO were developed for each subsystem for specific needs. There are some completed models which are used for the design and diagnostics. Since these modeling works were done independently, hardly any compatibility exists among those codes. Also each of these models was targetted for a specific design, and, therefore, often it is not easy to expand these for a new design.

So far, softwares were developed using either low level languages, like C or Fortran, or high level languages, like Matlab or Mathematica. A high level language, like Matlab, can make the prototyping very easy as compared to the low level language environment due to its well designed front end. Unfortunately, the overhead of such an environment is too heavy for the time domain simulation of the optics system. SMAC, for example, has the control system built into the Fortran code , whereas Matlab is used as the front end.

The End to End modelling effort was initiated in order to write one simulation program which includes all important subsystems for LIGO and for other similar experiments. One simulation code will be used for all purposes. It should be the time domain model so that it can simulate LIGO detector as close as possible to the reality, and it can generate realistic pseudo data which can be fed into the diagnostics and/or data analysis softwares.

The End to End simulation environment should be easily expandable so that one would be able to easily add new functionality and it would be possible to easily configure it for new or modified configurations. The modular approach should allow to develop models in modular way and combine them together later. E.g., SEI/SUS system, PSL and COC should be, and indeed are, developed independently and can be combined later. This is achieved by using object oriented design developed in LIGO using C++. This design and software package is called **Adlib**, “Adlib, the DigitaL Instrument Builder”.

The End to End simulation package has two layered structure. In the lower level, the program is built using Adlib written in C++. This program knows about pieces of hardwares, like mirrors and

LIGO DRAFT

digital filters, and it knows how they interact with each other, when connected. But it has no information hardcoded in about the actual connections or setups of hardwares.

When one runs this program, it reads in a file which describes the configuration of the system to be simulated, referred to as "**description** file" hereafter. The system can be as simple as one digital filter or can be as complicated as the full LIGO setup. Then the program generates time series of data or frequency spectrum. This is the upper layer of the End to End program, where one prepares the description file and analyze the output. The description file is a simple text file which describes how hardware pieces are configured. There is a program which creates and modifies these description files using graphical user interface, named Alfi, "**AdLib Friendly Interface**".

This two layer structure allows users to simulate wide ranges of configurations even if s/he does not know low level languages, like C++, with the full power of the compiled code. E2E program is like Matlab. Matlab has many functions built in, but Matlab itself does not know anything what to calculate. The lower layer of the End to End model is fairly complete, except for the support of alignment degree of freedom which is now being worked out. But the development for the LIGO simulation has just started, because there is almost no description files for it.

The in-house development of the code allows us to optimize the code. We can check the code for bottlenecks of the CPU and memory usages and improve the performance, either by modifying the technology or sacrificing the generality to achieve the necessary performance. As is explained later, End to End model offers the fast prototyping like Matlab, and it does not suffer from the overhead. This allows us to develop the code quickly, even for those parts which are called repeatedly with very fine time step.

The End to End model is too late to be used as a design tool of the first LIGO. Right now, the End to End model development is scheduled so that the main features of the key subsystems are completed first, which may allow us to use this software for the diagnostics of the first LIGO. As is explained above, this does not mean that one cannot use the package for other simulations in shorter time scale. Most of the key hardwares will be ready in Adlib by the middle of 1998. It takes so long to develop a complete software for the simulation of LIGO because we need to build very complex systems, and the validation is always a time consuming task.

40m experiment is used for two purposes, (1) the validation of software components and (2) the prototype of LIGO system. The check, if a given formula is properly implemented, can be done in various ways without any real experiment. The validation of the simulation using the experiment is needed because we need (1) to test if the physics behind the formula is close enough to the real setup, (2) to test if there is anything we missed in the implementation and (3) to understand the accuracy of the simulation.

3.2. E2E environment

The End to End model simulation environment consists of several components, which, as a whole, is referred to as E2E environment in this document. The following components are included in it:

- Software libraries to write simulation programs,
- A program to create and modify description files using GUI (graphical user interface),
- Programs which generate time series data and frequency spectrum.

LIGO-DRAFT

Because of the two layer structure explained above, there are different kinds of usage of E2E environment. The users can be grouped into the following three kinds.

The "**End User**" is a user who runs the program using existing description files to generate the time series of data or frequency spectrum, and analyzes the output. S/he may change the settings of the configuration.

The "**System Designer**" is a user who builds description files for systems and subsystems using Alfi. This may be as simple as an electronic box which contains only one digital filter, or may be as complicated as the complete LIGO configuration. These description files are like macros, and can be used as building blocks for other systems.

The "**Software Programmer**" is a user who writes C++ codes and builds the program. S/he does this to add new capabilities to the package, e.g., to add new modules for new hardwares, or to modify the functionality of the package in order to improve the speed of the program.

3.3. Road map

- If you want to try out this program, go read xxx.
- If you want to construct a new subsystem using existing modules shown in Table 2: "Primitive Modules", read xxx.
- If you want to add a module to implement new functionality, read xxx.
- If you want to put a new analysis code, read xxx.

The E2E package comes with four core documents and supplementary documents. The four core documents are

- **Overview of End-to-End Model** (LIGO-T970193-xx-E)
- **Organization of End-to-End model** (LIGO-T970194-xx-E)
- **Code Reference for End-to-End Model** (LIGO-T970195-xx-E)
- **Physics of End-to-End Model** (LIGO-T970196-xx-E)

"**Overview of End-to-End Model**", this document, is for all users who want to use E2E package. It provides the overview of E2E environment, explains the structure of the software, contains tutorials showing how to use it and how to customize. This contains most of the essential information, and this can be used as a quick reference. The road map of this document is given later in this section.

The second document, "**Organization of End-to-End model**", is for System Designers and for Software Developers. This document provides all information about the program structures, syntax of the description files, how to write the modules and how to create your own application.

Third document, "**Code Reference for End-to-End Model**" is for Software Developers. This document contains the full description of the classes and member functions available in the package. The reference file in the html format, which contains important part of this document, is available in html format. This html will be almost always up-to-date.

LIGO DRAFT

The fourth document, "**Physics of End-to-End Model**", is for all users who wish to understand the physics ingredients in the model. Physics and formulas included in the program codes are explained in this document, or proper references will be quoted for details.

Road map of this document ???

3.4. Systems simulated

The End to End model will simulate the following subsystems of LIGO.

- ASC
- LSC
- PSL
- IOO
- COC
  - Single mode (scalar field) optics systems can be constructed. When short and long cavities coexist, like long FP and Michaelson or short MC, one can use a resumption cavity module for the shorter part and can accelerate the speed if appropriate.
  - Multi mode optics system is being implemented, as of January 1998.
- COS
- SUS/SEI
  - SUS/SEI are being developed

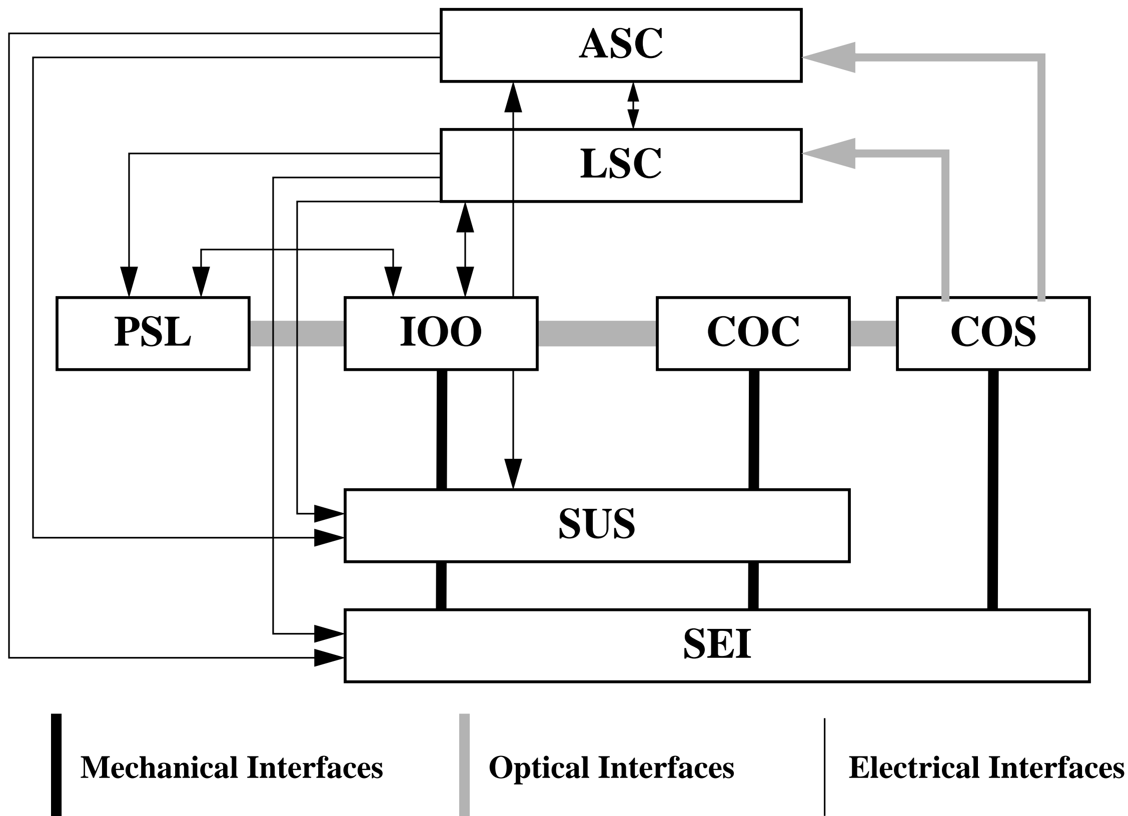


Figure 1: Subsystems and relations

- others

4 OVERVIEW OF THE E2E ENVIRONMENT

4.1. overview

Figure 2 shows the schematic view of the E2E environment. There are three major components of the E2E environment, (1) programs written using C++, (2) subsystem description files using special syntax created by a text editor or by a dedicated GUI editor, alfi,, and (3) the simulation run and analysis. The step-by-step explanation how to simulate is given in another chapter, “Simulation Work” in this note. In this chapter, the structure of the program is explained.

4.2. Useful things to know

4.2.1. Overview

This section summarizes key things one should know when working in the e2e environment.

4.2.2. Core optics related

4.2.2.1 Definition of length between optics

The length between mirrors are very important for the simulation of core optics. .

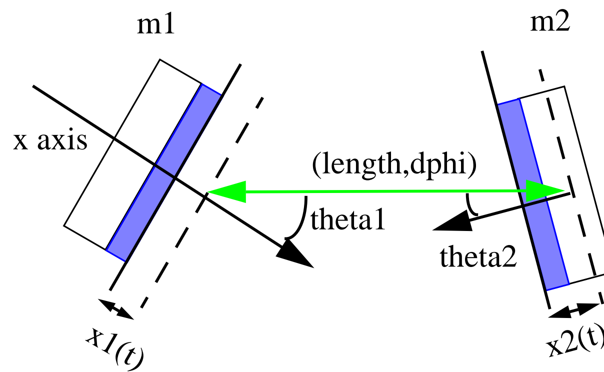


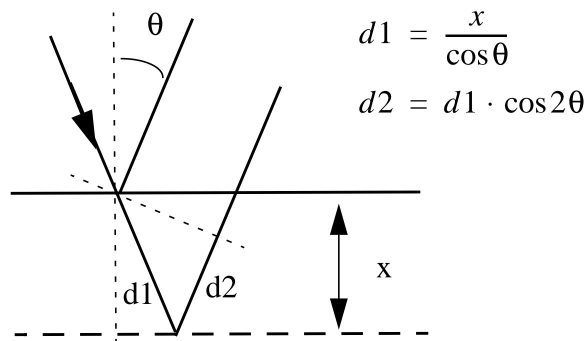
Figure 3: Definition of length

In Figure 3, the distance between the two mirrors, m1 and m2, are calculated using four quantities, length, dphi, x1(t) and x2(t). For each mirror, there is a reference place, shown using dashed lines in Figure 3, which is time independent. The distance between two reference planes is given by $L_0 = \left(N\left[\frac{length - lguoy00}{\lambda}\right] + \frac{dphi}{2\pi}\right) \cdot \lambda$, where N[x] is the nearest integer of x, and λ is the carrier wavelength of the field[1]. In other words, value of <macroscopic “length” minus “lguoy00”, the small length corresponding to the guoy phase acquired by the TEM00 mode in traversing this “length”> is rounded up to set equal to an integer multiple of the carrier wavelength, and the devi-

LIGO-DRAFT

ation from that is accounted for by dphi. With this convention, the numerical accuracy of the value used for “length” is not so important, and the distance can be set by the operational conditions, like the carrier being resonant. Without this scheme, one needs more than 13 digits to specify the 4km arm length in which the carrier field resonates.

The mirror position, which can be time dependent, is defined to be the relative distance between the mirror surface and the mirror reference plane, using the perpendicular direction pointing outward from the coated side (shown by a gray box in Figure 3) of the substrate as the axis. So in Figure 3, x1 is negative while x2 is positive. The effect of the mirror displacement, x1 and x2, are taken into account by the change of the phase. As is shown in Figure 4, the net change of the path



$$d1 = \frac{x}{\cos\theta}$$

$$d2 = d1 \cdot \cos 2\theta$$

Figure 4: Phase change due to displacement

length is $2x \cdot \cos\theta$, and the phase change due to this difference is added to the reflected field.

4.2.2.2 The reflection surface

In the model, the convention is used so that the sign of the field changes when reflected on the coated side, $E_{out} = -|r|\, E_{in}$. There are two settings which change orientation of a mirror, “theta” for a mirror and “dirX” for summation optics.

Setting value “theta” of a mirror is defined to be the incident angle of the field coming into the mirror, as is defined in Figure 3. When “theta” is larger than $90°$, the sign of r changes such that $E_{out} = |r|\, E_{in}$, because the field hits the coating from the substrate side. So the configurations (a) and (b) in Figure 5. represent the same configuration. Here (a) is a setup consisted of a pair of mirrors with the both coated sides inward just as the normal FP cavity, then “theta” is changed to π, while (b) is a configuration where the coated side of the second mirror is facing outward with “theta” = 0. (need clarification, suggestion ?)

Summation optics, as is explained later, are optics systems which have several mirrors included. There are two of this kind, summation cavity and summation michelson cavity. The optics config-

1. The wavelength of a field is defined as $\frac{2\pi}{\lambda} = \frac{2\pi}{\lambda_0} + \delta k$, using a reference wavelength λ_0. For the carrier field, The term dk is 0 for the carrier, and $\pm\frac{2\pi}{\lambda_{RF}}$ for the first sidebands when noises are neglected. Strictly speaking, λ in the definition of L_0 is the reference wavelength.

LIGO-DRAFT

uration for them are shown in Figure 5. There is no parameter “theta” for the mirrors in the summation optic. In stead, there is a parameter called dirX. If dirX is positive, the coating of the mirror is facing as is shown in Figure 5, while if dirX is negative, the coating side is changed. E.g., if dirB = -1 and dirC = -1 in a summation michelson cavity, all mirrors have the coating facing toward inside.

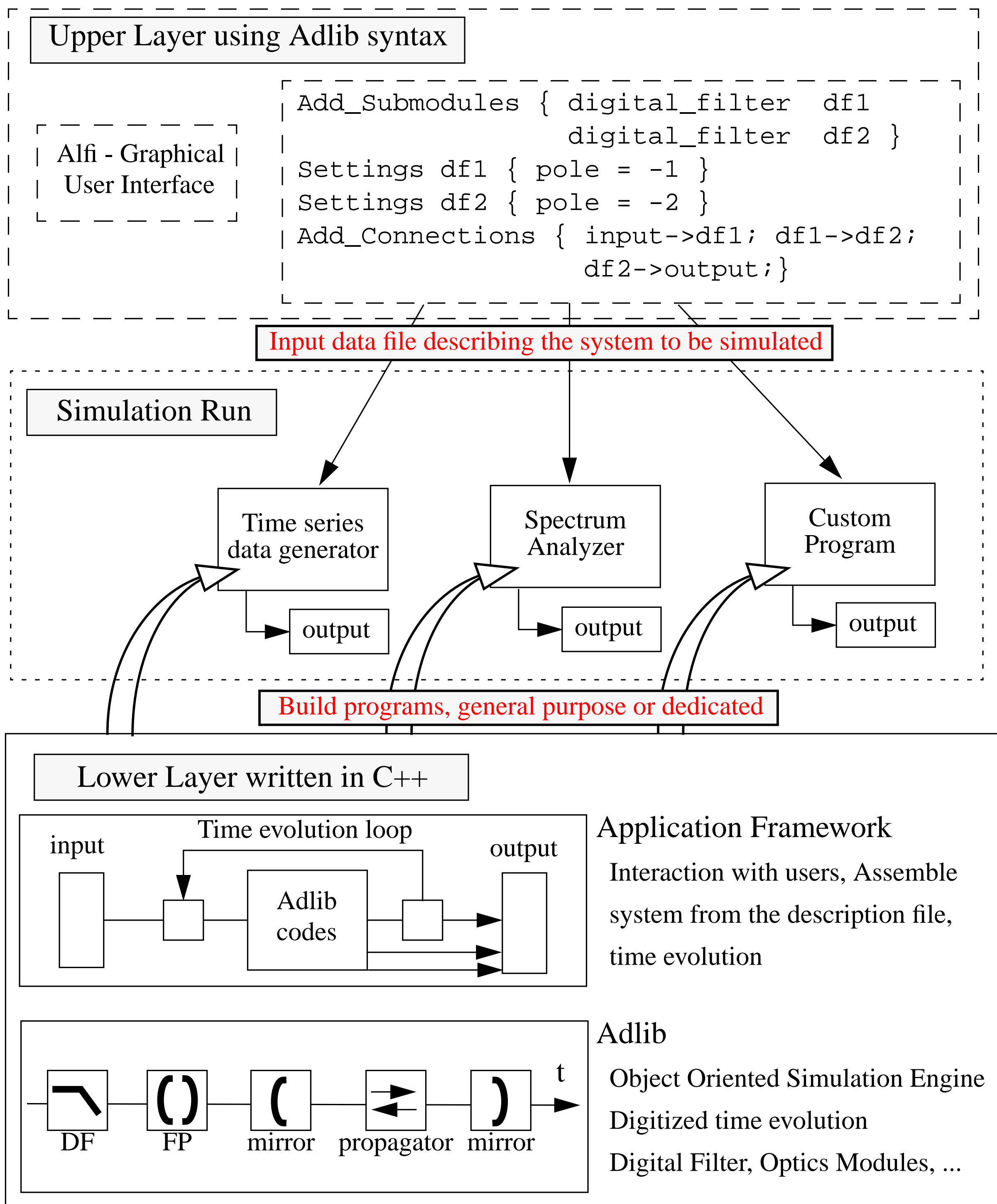


Figure 2: Schematic view of the End to End model

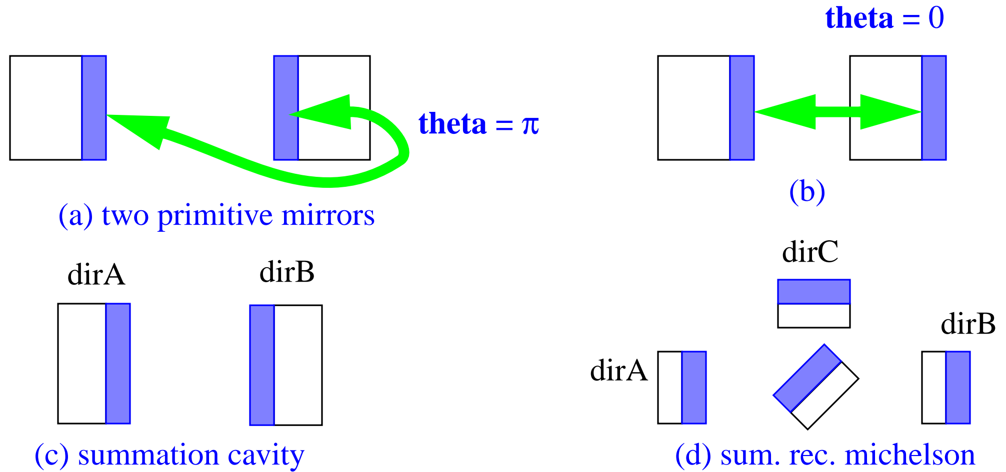


Figure 5: Direction of coating

4.2.2.3 Cavity simulation with a odd time step - or how to simulate an asymmetric Michelson

If the time step is the one-way trip time of a cavity, τ_0, or integer faction of τ_0, the program can do the simulation without approximation. When there are several cavities involved, like asymmetric Michelson cavity, one cannot choose an optimal time step for both. E.g., if the inline arm length l_{in} is 10m and the offline arm length is 10.2m, the idealistic time step is 0.2m/c, rather than 10m/c, and the simulation takes 50 times longer. This is taken care in the following way, so that one can use a time step not exactly an integer fraction of one trip time of a cavity.

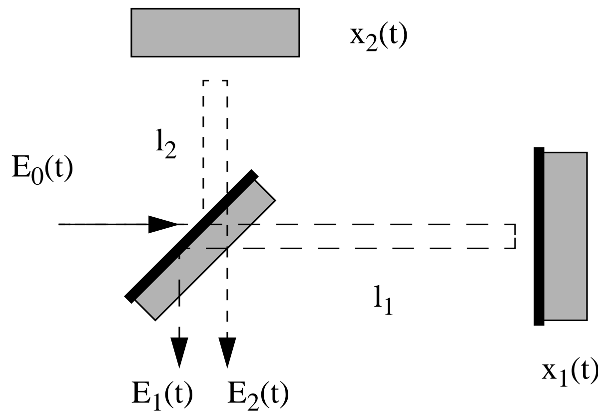


Figure 6: Asymmetric Michelson cavity

The dark port output fields $E_1(t)$ and $E_2(t)$ can be written using $E_0(t)$ as follows, where r is the reflectance of the end mirrors, and, for simplicity, the transmittance and reflectivity of the beam splitter are omitted. In this equation, $\tilde{E}(t)$ is the slowly changing part of the field defined as $E(t) = \tilde{E}(t)\exp(i\omega t)$.

LIGO-DRAFT

$$\begin{aligned}
E_i(t) &= r(t-\tau_i)E_0(t-2\tau_i) \\
&= r(t-\tau_i)\tilde{E}_0(t-2\tau_i)\exp(i\omega(t-2\tau_i)) \\
&= \exp(i\omega t)r(t-\tau_i)\tilde{E}_0(t-2\tau_i)\exp(-i2kl_i) \\
&= \exp(i\omega t)\wp_i[r(t-\tau_0)\wp_i[\tilde{E}_0(t-2\tau_0)]]
\end{aligned}$$

$$\wp_i[f(t)] = \exp(-ikl_i)f(t-\delta\tau_i) \approx \exp(-ikl_i)\left(f(t)-\delta\tau_i\frac{df}{dt}\right)$$

$$\tau_i = \frac{l_i}{c} \qquad \delta\tau_i = \tau_i - \tau_0$$

The rule derived from this formula is as follows. For each propagator, (1) calculate the field entering to the propagator at digitized times (n τ_0), and (2) calculate the field at the end of the propagation by correcting it for the difference of the times of the correct trip time and the discretized travel time and by multiplying the real phase factor, exp(-ikl$_i$). For the mirror, calculate the reflection at the digitized time.

4.3. Lower layer using C++

4.3.1. Overview

The simulation program is written in C++.

4.3.2. Adlib - Object Oriented Time domain simulation engine

Modify Adlib

- Simulation engine
- time domain
- C++
- OO
- speed optimization
- digital filter
- digitized time evolution
- modular and expandable
- improvement for speed
- new capabilities

Primitive Module Development

- Optics with alignment

LIGO-DRAFT

- mpm linear optics

Modules

Time evolution

4.3.3. Application Framework

Interaction with users

Construct the system to simulate for Adlib engine from the description file.

Data I/O

Time loop

4.4. upper layer using Adlib syntax - Alfi, GUI front-end

Adlib syntax

GUI - Alfi

4.5. simulation run

How to do the analysis

Time series data and Frequency spectrum, application provided

Custom program

Modify the application framework,

write new modules

5 SIMULATION WORK

5.1. Overview

The simulation work is consisted of three steps, (1) modifying the source code and building a program, (2) build description files for the system to be simulated and (3) run the program with the description file and analyze the result. We will go through each step.

5.2. Modify C++ code - Software Programmer

5.2.1. When you need to modify the source code

Because fairly complicated tasks can be done in the upper layer, you will not need to modify the source code so often. The modification of the C++ source code is needed when

LIGO-DRAFT

- you need to add a new capabilities
- you want to insert a special analysis tools
- you want to build a new general purpose program
- you want to open the Pandora's box (explained in "**Organization of E2E**" and "**Code Reference for E2E**")

Adlib has many simulation tools included which simulate commonly used hardwares, e.g., a pockel cell modulating the field either by phase modulation, which is very slow to simulate but close to reality, or by adding sidebands, which is fast to simulate but an approximation. When you want to simulate hardwares not supported by Adlib or when you want to improve the sophistication of the modeling, you need to add or modify the code.

Another case is when you need to do some special analysis and have to place that routine in the hardware setup. E.g., you may want to save precise information when spurious event occurs and keep logging for a given time and terminate. In this case, you will write a module and plug that module into the place where you can catch the spurious event. In this case, you add a module and modify the description file as well.

In these two cases, you write a module following the guideline below, and make one change in the main program, and build a program using a Makefile, which is explained in other document.

A good example of the third case is the spectrum analyzer. This program which generates the frequency spectrum is a variant of the program which generates the time series of data. By modifying the main program, you can customize the input and output for your needs. The understanding of this mechanism will help to incorporate the second kind modification listed above.

5.2.2. How to build a program

The main program which generates the time series of data is as follows:

file "modeler_main.cc"

```
#include  "modeler_base.h"
int main(int argc, char** argv);
int main(int argc, char** argv)
{
   modeler_base                modeler;
   modeler.run(argc, argv);
}
```

modeler_base is a base class of the application framework, and you modify only those member functions you need to customize.

The structure of the modeler_base class is as follows (only important parts are retained, the source files, modeler_base.h and modeler_base.cc contains full information, and the detail explanation is provided in "**Code Reference for E2E**"):

file "modeler_base.cc"

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

LIGO-DRAFT

```
// ----------------------------------------------------------------------------
//  pseudo main program
void modeler_base::run( int argc, char *argv[] )
{
// Handle options (-help, -v, -V, -d0~d4 are handled)
   if ( !handle_options( argc, argv, checkit ) ) exit(1);
//  print information about this program, purpose, version, main author, etc
    about();
//  global initialization
    global_initialize();
// read main description file and connect data lines
    if ( !construct_system() ) exit(1);
// get the parameter file and set parameters
    if ( !parm_file_parser(parm_file, in_list, true) ) exit(1);
// miscellaneous initialization, if needed
    if ( !misc_init() ) exit(0);
//  time evolution simulation and analysis
    main_loop();
}
// ----------------------------------------------------------------------------
//    * construct_system
// ----------------------------------------------------------------------------
//  parse the main description file and construct internal structure for process
bool modeler_base::construct_system( void )
{
// setup I/O interface
    if( !setup_io_interface() ) return false;
// register available types
    if( !register_types() ) return false;
//  build simulation from root box descriptor file
//  makes connections and checks validity of input/output names
//  for all internal modules (does not check input/output type validity)
    if ( !build_the_model( ) ) return false;
//  build external output list
    if ( !setup_external_output() ) return false;
//  build external input list
    if ( !setup_external_input() ) return false;
//  check internal data type validity and consistency
    if ( !type_check() ) return false;
//  build queues by dynamic compilation of links
```

LIGO-DRAFT

```
    build_lists();
    return true;
}
// ---------------------------------------------------------------------------
//    * register_types
// ---------------------------------------------------------------------------
//  register the types available in the model
bool modeler_base::register_types( void )
{
// register module data_in
   the_model.add_submodule_type(data_in());
...
...for all built-in modules xxx
...the_model.add_submodule_type(xxx())
...
   return true;
}
// ---------------------------------------------------------------------------
//    * main_loop
// ---------------------------------------------------------------------------
//  time evolution loop
void modeler_base::main_loop( void )
{
//  change paramneters and repeat run until done
  while( init_parameters( ) )
  {
//  major time loop
//  evolution_continue tests if the time evoultion should still continue
//  the modeler_base.evolution_continue() tests if the time has reached
//  the end time user specified.
     while( evolution_continue() )
     {
//  any analysis of the state before the time is incremented
       if( !analyze_pre_state( mw ) ) break;
//  time is incremented and all modules are called in the linked order
       process_evolution();
//  any analysis after the time evolution
       if( !analyze_post_state( mw ) ) break;
     }
//  decide if keep running with new parameters
```

LIGO-DRAFT

```
            if( !params_continue( mw ) ) break;
        }
    }
```

These are major member functions of the base class modeler_base, **run**, **construct_system** and **main_loop**. Those functions shown in bold face are virtual functions, and you can override to suit for your needs. When you create a new module, only modification needed is to override register_types and call add_submodule_type. The example will be given in the following section.

The main routine of the class is ***run***, as is clear from the code of *modeler_main.cc* shown above. It initializes, construct data flows, and handle time loop. When the data flow is constructed in **construct_system**, the modules in the chain are placed in queue, and those modules in the queue are called in sequence in **main_loop.**

Functions to be included soon

- message passing to arbitrary module
- Status dump and reload, so that a simulation can be interrupted and continued later
- parameter input to the main program should support Root.box1.box2.box3.val1 = 1.0

5.2.3. How to write a module

5.2.3.1 What is a module

The system to be simulated is constructed by combining building blocks called modules. Each module may represent an object like a mirror, an electric device like lazer or photo diode, input signal, like a sinusoidal signal or a time series of seismic motion, or an output to a text file or a pipe to other program. The module can also be an analysis program you invent. The fidelity of the modules ranges from high to low. Some simulate the reality very precisely, while some very crudely. Fidelity depends on the process to be simulated. Some effect may not be important for most of the field calculation, but it may be crucial when you are tracing a tiny noise.

One special module called “box” can contains other modules. E.g., Figure 9 shows a box which contains several modules and several input and output ports. Creating a box is like write a function with arguments. Once created, the box module can be used just as primitive modules which are coded in Adlib library (see Table 2: "Primitive Modules").

5.2.3.2 When do you write a module

In order to support a new device or to do special analysis, you have to write a new module and include it in the description file. Analysis can also be included in the main program as is explained in 5.2.2. "How to build a program" if you are analyzing the data coming from a well defined port. But if you are going to insert an analysis probe in a complicated system, it is easier to write a module and insert the module in the description file.

LIGO-DRAFT

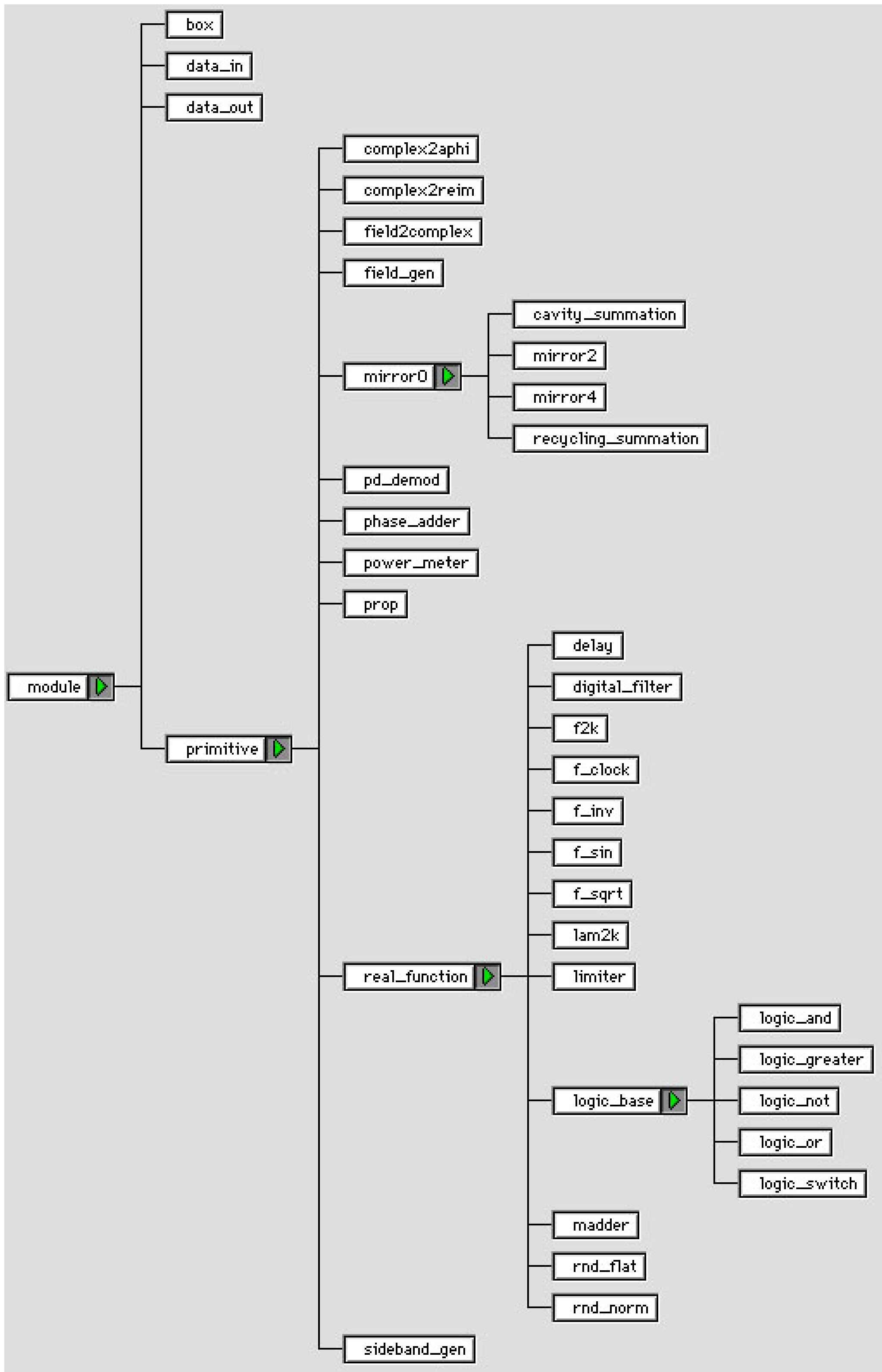


Figure 7: Class Hierarchy of modules

LIGO DRAFT

5.2.3.3 module class

In Adlib, all modules are derived from a base class “module”. The class hierarchy related to the modules are shown in Figure 7.

5.3. System Designer

Compound module development

- Construction of subsystems
- Analysis probe
- Setup of simulation subsystems - SUS/SEI etc

Validation

5.4. End User

Change parameters of existing box for subsystems

5.5. General

5.5.1. GUI - Alfi

- Make it easy to construct, modify and maintain the description file
- Future - interactive change of setting
- integrated data visualization

5.5.2. g++ and ANSI

5.5.3. DOC++

6 USING THE PROGRAM - STEP BY STEP

6.1. Overview

LIGO-DRAFT

6.2. Data types and existing modules

Table 1: "Data types" summarizes data types used in Adlib, defining settings for modules and passing data between modules. “type name” is the name used for the documentation purpose,

Table 1: Data types

type name	*description*	*example*	*data type*
complex		zeros and poles of digital filter	adlib_complex
integer		number of side-bands of field	int
real		reflectance of mirrors	adlib_real
single_mode_field		input and output of optics objects	smfield
string		type specification of data_in	string
unknown	data type assigned to a port whose data type is determined by other conditions, like the output port of data_in which is determined by the “type” setting.	output of data_in	N/A

Table 2: Primitive Modules

Name	*Function*	*in*	*out*	*setting*
I/O				
data_in	used to get data into the simulation	none	“0” variable type	"type" string ("real"), "init" output type (???)
data_out	used to get data out of the simulation (a "probe")	"0" variable type	none	none
Real Function				
madder	implements z = a*x + b*y	"a" "x" "b" "y" real	"0" real	none
sine	the sine function	"0" "amplitude" "frequency" "phase" real	"0" real	none
square_root	the square root function	"0" real	"0" real	none
inverse	the inverse function	"0" real	"0" real	none
digital_filter	a digital filter	"0" real	"0" real	"zero" "pole" "gain" real "zeropair" "polepair" complex
limiter	models a circuit with rails	"0" "upper" "lower" real	"0" real	none
Logic functions **Input "val" is evaluated to be true if val > threshold, otherwise false. Output is true_val if the result is logical true, false_val otherwise.**				

LIGO-DRAFT

Table 2: Primitive Modules

Name	*Function*	*in*	*out*	*setting*
and	logical AND	"a","b" real	"0" real	"threshold" (0.9), "true_val" (5), "false_val" (0.0) real
or	logical OR	"a","b" real	"0" real	same as above
a>b	comparison	"a","b" real	"0" real	same as above
not	negation	"0" real	"0" real	same as above
switch	if bool is true, return high, else return low.	"bool" "low" "high" real	"0" real	same as above
Data Generation				
rnd_flat	generates random numbers with a flat distribution	"range" real	"0" real	none
rnd_norm	generates random numbers with a normal distribution	"width" real	"0" real	none
clock	generates the time	none	"0" real	none
Unit Conversion				
lam2k	converts wavelength to wavenumber	"0" real	"0" real	none
f2k	converts frequency to wavenumber	"0" real	"0" real	none
Type Conversion				
field2complex	converts a field to a complex number	"0" single_mode_field, "dk" real	"0" complex	none
complex2reim	converts a complex number to real and imaginary	"0" complex, "phi" real	"real" "imag" real	none
complex2aphi	converts a complex number to amplitude and phase	"0" complex	"amp" "phi" real	none
Field Operation				
field_gen	generates a field	"power" "phi" real	"0" single_mode_field	"lambda" real (1.0)
sideband_gen	phase modulates a field (uses sideband approximation)	"0" single_mode_field, "k_mod" "gamma" real, "order" integer	"0" single_mode_field	none
phase_adder	phase modulates a field directly	"0" single_mode_field, "phi" real	"0" single_mode_field	none
power_meter	outputs the power in a field	"0" single_mode_field	"0" real	none
pd_demod	demodulates a field	"0" single_mode_field, "k_demod" real	"demod" complex, "power" real	none
Optics				

LIGO-DRAFT

Table 2: Primitive Modules

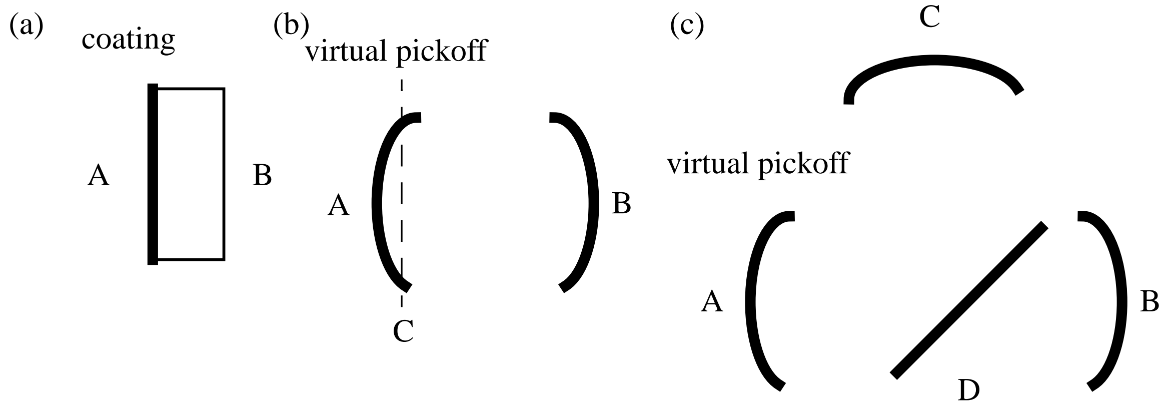
Name	*Function*	*in*	*out*	*setting*
prop	propagates a field over a macroscopic distance	"0" single_mode_field	"0" single_mode_field	"length" real (1.0) "dphi" real (0.0) (f)
mirror2 (a)	a 2-input 2-output mirror (cavity end mirror)	"x" real "Ain" "Bin" single_mode_field	"Aout" "Bout" single_mode_field	"r" "t" "R" "T" "L" real (2.0), (d) "angle" real (0.0)
mirror4 void don't use	a 4-input 4-output mirror (beam splitter, pickoff)	"x" real "A+in" "A-in" "B+in" "B-in" single_mode_field	"A+out" "A-out" "B+out" "B-out" single_mode_field	"r" "t" "R" "T" "L" real (2.0), (d),(e) "angle" real (M_PI/4.0)
Summation Optics:				
cav_sum (b)	represents a short FP cavity	"xA" "xB" "xC" real, "Ain" "Bin" single_mode_field	"Aout" "Bout" "Cout" single_mode_field	"length" real (1.0), "dphi" real (0.0) "dirA" real (1.0), "dirB" real (1.0) "rA" "tA" "RA" "TA" "LA" real (2.0), (d) "rB" "tB" "RB" "TB" "LB" real (2.0), "rC" "tC" "RC" "TC" "LC" real (2.0)
rec_sum (c)	represents a recycled MIFO	"xA" "xB" "xC" "xD" real, "Ain" "Bin" "Cin" "Din" single_mode_field	"Aout" "Bout" "Cout" "Dout" "Bpick" "Cpick" "Dpick" single_mode_field	"lengthA","lengthB","lengthC" real (1.0), "dphiA","dphiB","dphiC" real (0.0) "dirA","dirB","dirC","dirD" real (1.0) "rA" "tA" "RA" "TA" "LA" real (2.0), (d) "rB" "tB" "RB" "TB" "LB" real (2.0), "rC" "tC" "RC" "TC" "LC" real (2.0), "rD" "tD" "RD" "TD" "LD" real (2.0),

while "data type" is the name used in the C++ code. The real variables are refered to using "adlib_real" as the data type as much as possible, so that it would be easy to switch to different byte sizes. "adlib_complex" and "smfield" also use adlib_real for the real variable. The default is double type.

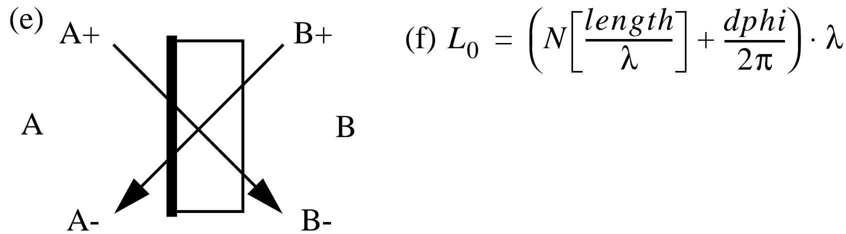
Some modules to be added soon

- function parser
- simple logic switch

LIGO-DRAFT



(d) Any two of the R, T, L (power reflectance, transmittance and loss), r, t, l (amplitude) can be specified for a mirror.



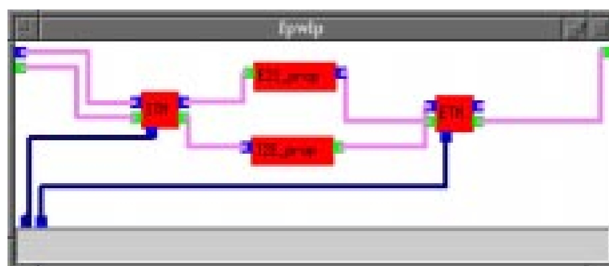
(f) $L_0 = \left(N\left[\frac{length}{\lambda}\right] + \frac{dphi}{2\pi}\right) \cdot \lambda$

6.3. Box syntax

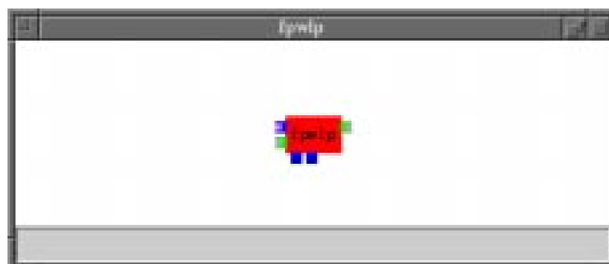
6.3.1. An Example

The code shown below, FP.box, is an example description file consisted of two mirrors:

LIGO-DRAFT



Internal view



External view

Figure 9: FP box by alfi

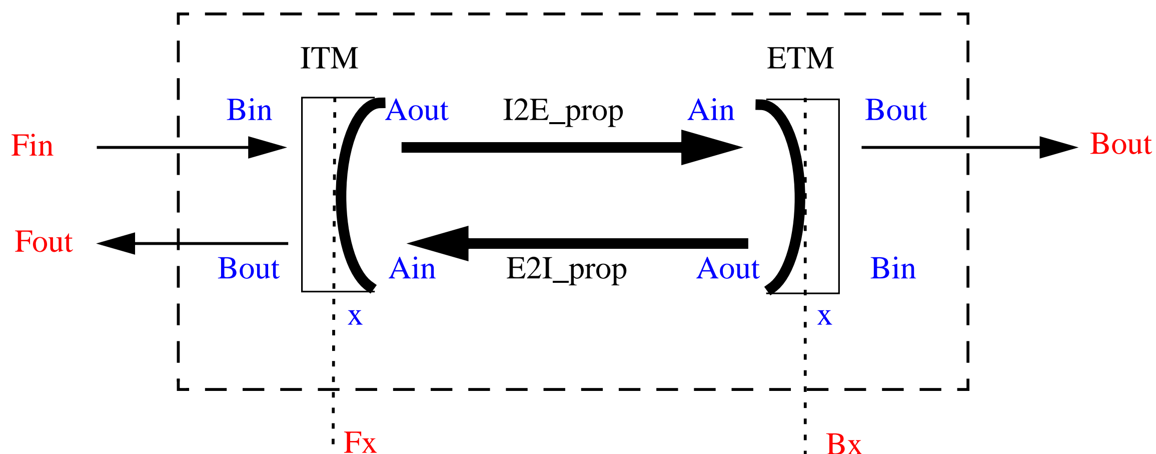


Figure 8: Schematic view of FP box

Mostly, you will use Alfi, the GUI program, and you do not need to understand the syntax how to describe the system. The alfi windows creating the FP box is shown in the following figures. There are two mirrors, two propagators and several input and output ports, and they are connected by linking ports by mouse. But the syntax is very easy, and the understanding of the syntax may help you when you are constructing the system, and will help you when you want to make a minor change, like changing just a few parameters.

LIGO DR

file FP.box

```
Add_Submodules                  % declare modules used in this file
{
  prop I2E_prop                 % one propagator named I2E_prop
  prop E2I_prop                 % another propagator named E2I_prop
  mirror2 ITM                   % one mirror named ITM
  mirror2 ETM                   % another mirror named ETM
}
Settings ITM                    % setting for input test mass
{
  %Input test mass
  R = 0.97                      % power reflectance
  L = 50e-6                     % power loss
}
Settings ETM
{
  %End Test Mass
  T = 10e-6                     % power transmittance
  L = 50e-6                     % power loss
}
Settings I2E_prop               % define setting for I2E_prop
{
  %LIGO Arm length              % from % to the end of the list is a comment
  #include length.param         % inset the content of length.param
}
Settings E2I_prop               % same for E2I_prop, propagator from ETM to ITM
{
  %LIGO Arm Length
  #include length.param         % inset the content of length.param
}
Add_Connections                 % Connection, or time flow, between modules
{
  this Fin -> ITM Bin           % input field goes to ITM input
  ITM Aout -> I2E_prop 0        % transmitted field goes to propagator
  I2E_prop 0 -> ETM Ain         % propagator is terminated at ETM
  ETM Aout -> E2I_prop 0        % reflected field goes to propagator
  E2I_prop 0 -> ITM Ain         % this progagator goes to ITM
  ITM Bout -> this Fout         % reflected field goes to Fout
  ETM Bout -> this Bout         % transmitted light through ETM goes to Bout
  this Fx -> ITM x              % Fx determines ITM x position
```

LIGO DRAFT

```
  this Bx -> ETM x             % Bx determines ETM x position
}
```

file length.param

```
length = 4000                  % setting for the propagator
```

6.3.2. Declarations - Add_Submodules, Settings and Add_Connections

The description file is consisted of three declarations, “Add_submodules”, “Settings” and “Add_Connections”. Compare this code with the following C code.

```
int i. j;
i = 1;
j = i;
```

As can be seen, “Add_submodules” corresponds to the declaration of variables, “Settings” to the initialization of the variable, and “Add_Connections” to the operation among variables, sort of.

Creating a description file is like creating a function or macro with arguments. Now let us go through the above example, *FP.box*. By the way, from “%” to the end of the line is discarded when this file is parsed, i.e., you can place comments with % at the head. “%*” is used by Alfi to place auxiliary information.

First, you have to declare what kind of modules, be primitive or other description file you created, are used in this module, and how they are named. In this example, there are two mirrors, named ITM and ETM, and two field propagators named I2E_prop and E2I_prop.

Then follows the “Settings” for each module. Here, you give values for those quantities which do not change during the simulation. For the mirrors, you specify reflectance, transmittance and/or losses. You specify any of the independent two quantities, and the rest are calculated. As is used for the I2E_prop and E2I_prop, you can include the content of a file by using the “**#include**” declaration. In this example,

```
Settings I2E_prop              % define setting for I2E_prop
{
  %LIGO Arm length             % from % to the end of the list is a comment
  #include length.param        % inset the content of length.param
}
```

is the same as

```
Settings I2E_prop              % define setting for I2E_prop
{
  %LIGO Arm length             % from % to the end of the list is a comment
  length = 4000                % setting for the propagator
}
```

when the content of the file length.param is

LIGO-DRAFT

```
length = 4000                  % setting for the propagator
```

This length of the propagator defines the distance between the two mirrors. Strictly speaking, this length is microscopically adjusted so that the carrier resonates in this cavity. The deviation from the resonance, or any dynamical motion of the mirror, is handled by the input parameter of the mirror, “x”.

The last declaration in the file is “Add_Connections”. In this declaration, you define the flow of data. In the above example, input field, external parameter Fin, is connected to the non-coated side of the ITM, then the field coming out from the coated side of ITM is connected to the propagator I2T_prop, which propagates the field from ITM to ETM. This propagator is connected to the coated side of the ETM, and the reflected field is connected to the propagator which propagates the field from ETM to ITM. The reflected field coming out of the non-coated side of ITM is connected to the external parameter Fout. The transmitted light through ETM is connected to the external parameter Bout. The positions of the mirrors are given by the two external parameters, “Fx” and “Bx”.

6.3.3. See details in other reference

There are a few more declarations which are extensively used by GUI. Those are fully explained by “**Organization of End-to-End model**”.

6.4. How to analyze

Application framework

Time series

Spectrum Analyzer

User defined

6.5. How to write modules

Overview of module related classes

On-line documentation

How to write - examples

6.6. Source code maintenance

6.6.1. Organization

LIGO-DRAFT

6.6.2. maintenance

CVS

DOC++

7 SUMMARY

xxx

APPENDIX 1 APPENDIXTITLE

xxx

APPENDIX 2 ANOTHER APPENDIX

xxx

LIGO-DRAFT