

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

<b>Document Type</b> <b>LIGO-T980051-00 - E</b> June. 1998
<b>Getting Started with End-to-End model</b>
Biplab Bhawal, Matt Evans, Ed 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 KEYWORDS

Simulation, End-to-End model, time-domain, time-series, frequency-domain, frequency-series, spectrum, Fabry-Perot cavity, Digital Filter, thermal noise, modules,

# 2 DEFINITION OF WORDS

Adlib	Adlib : A Digital Instrument Builder - C++ simulation engine used in e2e
Alfi	AdLib Friendly Interface - GUI front end of e2e
CVS	Concurrent Version System
e2e or E2E	End to End model
egcs	Free gnu compiler package, including C++,C,Fortran, available at egcs.cygnus.com.
MATLAB	software package developed by The Mathworks Inc., MA.
Mathematica	software package developed by Wolfram Research, Inc., IL
SMAC	Single Mode Acquisition Code (developed at Caltech)
Twiddle	Mathematica package for freq-domain optics analysis(developed at Caltech)

# 3 ABSTRACT

This document is intended to help you start using the End to End model by going through simple examples step by step. Appendix Appendix 1 summarizes the overall structure of the End to End package. This tutorial includes

- how to setup your computer to use the End to End model,
- how to run e2e to generate time series of the data and frequency spectrum,
- how to analyze the data, and
- how to write a simple module to add a new functionality to e2e.

The main purpose of this document is to go through all of these key steps as quickly as possible to let you have an idea what e2e is and let you judge yourself how e2e can be used. The details of the e2e environment are provided in the e2e core documents:

- **Overview of End-to-End Model** (LIGO-T970193)
- **Organization of End-to-End model** (LIGO-T970194)
- **Code Reference for End-to-End Model** (LIGO-T970195)
- **Physics of End-to-End Model** (LIGO-T970196)
- **Extending E2E Model - The How to Guide to Coding Modules** (LIGO-T980067)
- **Updates to the multi-mode version of End-to-End model** (LIGO--T990004)
- **Alfi - the GUI of the End-to-End Model** (LIGO-T980014)

The main components of e2e are (1) the time domain simulation engine named Adlib, and (2) the front-end to configure the setup you are going to simulate, namely Alfi. The first six documents describe Adlib, an application framework and the physics behind, and the last document is the manual for Alfi, which provides the Graphical User Interface front-end of the simulation package.

This document is organized as follows. In Sec. 4, the purpose and functionality of the End to End model are briefly explained. Sec. 5 summarizes what you have to do before you start this tutorial. Sec. 6 explains how to build a setup you want to simulate, followed by Sec. 7 which explains how to run e2e to simulate the setup you prepared. The output format of the data is discussed also in Sec. 7. Sec. 8 briefly explains how to add a new functionality to the e2e environment. Examples using C++, C and Fortran are given in a separate document (LIGO-T980067).

## 4 WHAT IS END TO END MODEL

End to End model is a program package designed to simulate the LIGO interferometer in the time domain. The motivation and the main purpose of e2e are given in “**Overview of End-to-End Model**”. The underlying design is made so that

- new functionality can be easily added, and
- the program can be setup easily to simulate wide range of systems.

Because of this design, this can be used for many different purposes.

The e2e is not a replacement of matlab or like, but rather a complement. The strength of these packages is the generality of their use. The high level language is flexible enough so that many kind of calculations can be done in a much easier way than by using software written in low level languages, like C++. On the other hand, this “high level generality” introduces the overhead in the performance. This prohibited SMAC programmers to use the matlab controls in conjunction with the optics code written in fortran and implemented as a mexfile. The control system had to be written using fortran. The e2e does not have this kind of generality, but is designed to avoid this kind of overhead.

## 5 PREPARING FOR E2E

The e2e simulation package has several programs and files in it. To let it work, you need to define some environmental variables summarized in the following table. PATH and LD\_LIBRARY\_PATH should be appended to the existing paths, and CVSROOT and E2E\_MAKEFILE\_CFG are needed when you want to add a new feature to e2e.

One example of the setup under C shell is given below. Copy those lines in your .cshrc file for permanent use. If this does not work, ask your system manager.

```
setenv E2E_SOFT_HOME      /home/e2e/Software
setenv PATH  ${E2E_SOFT_HOME}/bin/:${PATH}
setenv LD_LIBRARY_PATH  ${E2E_SOFT_HOME}/lib:${LD_LIBRARY_PATH}
setenv E2E_PATH  .:${E2E_SOFT_HOME}/lib
setenv E2E_LD_PATH  ${E2E_SOFT_HOME}/lib/modules
setenv CVSROOT  /home/e2e/Software/cvsroot
setenv E2E_MAKEFILE_CFG  /home/e2e/Software/config/Makefile.cf
```

**Table 1: Environmental Variables on CIT-LIGO network**

<i>purpose</i>	<i>variable name</i>	<i>Value</i>
<i>All</i>	E2E_SOFT_HOME	/home/e2e/Software
	PATH	\${E2E_SOFT_HOME}/bin (append)
	LD_LIBRARY_PATH	\${E2E_SOFT_HOME}/lib (append)
	E2E_PATH	.\${E2E_SOFT_HOME}/lib
	E2E_LD_PATH	\${E2E_SOFT_HOME}/lib/modules
<i>code development</i>	CVSROOT	\${E2E_SOFT_HOME}/cvsroot
	E2E_MAKEFILE_CFG	\${E2E_SOFT_HOME}/config/Makefile.cf

You don't need any programming, in order to do the simulation using the existing functionality of e2e ( Sec. 6 to Sec. 7 in this tutorial ). If you wish to add a new functionality, or if you want to customize the output ( Sec. 8 ), you will need compilers. The software is developed using egcs, free compilers including C++, C and FORTRAN (each called g++, gcc and g77, respectively). C++ compilers are now moving to the ANSI standard, and g++ compiler is close to that goal (but not quite). Other compilers, including SUN CC C++ compiler, may fail to compile e2e code because their implementation is not current. So, use g++, gcc and f77 for C++, C and FORTRAN compilers to avoid compatibility problems.

## 6 DEFINING WHAT TO SIMULATE

### 6.1. What is this step for

In order to do the simulation work using e2e, the first thing you have to do is to describe the setup you want to simulate. This step is just like what you write a source code in C:

```
double foo( int in_i, double in_d )
{
    double ddd;
    d2 = exp(in_i) + sin(in_d);
    return ddd;
}
```

Instead of using a text editor to write the C source code, you use the program “alfi”, part of the e2e package, to write a file describing the setup, called a description file, using a GUI (graphical user interface). Instead of using built-in C functions, like sin or exp, you use primitive modules like “**field\_gen**”, which generates a monochromatic laser, or “**sideband\_gen**” which adds a sideband to the incoming laser. Just as C functions have input parameters of various types and return function values, you can define input and output ports of various predefined types to your description file.

And just as your C function can call other functions you defined, one description file can use other description files.

In this tutorial, a simulation will be setup, step-by-step, to generate a simple thermal noise,  $f^{-1/2}$  tail plus a resonant peak. Also provided is a set of files which can be used to calculate responses of a Fabry-Perot cavity.

## 6.2. Alfi - the front end of the simulation engine

*Note: Before we start this section, we wish to tell you that the illustrating figures in this document may differ slightly from what you'd actually see on your screen while running Alfi, our graphical user interface. Alfi is still adding new features and we're trying to accomodate more and more requests from users at this point of time. We try to keep this document updated, but, in case you find any differences, please be assured that such differences would not be a major hindrance to the first several steps of your learning process.*

The details of this software is given in the Alfi manual (T980014). In the windows shown below, all texts written in italic and arrows are comments, and will not exist in a real window. Once you complete your simulation setup as explained in Section 5, just type `alfi3` (the current version is 3) in the shell window to run the program. You will see the main window in Figure 1. Select “Box” from the “File” menu. A new window, “Box selection window” comes up. In this window, you create a new description file - called box file for short - or open an exiting one. Type `f_half` for the name and click OK. Another window, Box window, comes up. Choose “Save” from the “File” menu in the main window. Congratulations, you have written a program using alfi. Make sure that there is a new file created in your directory named `f_half.box`.

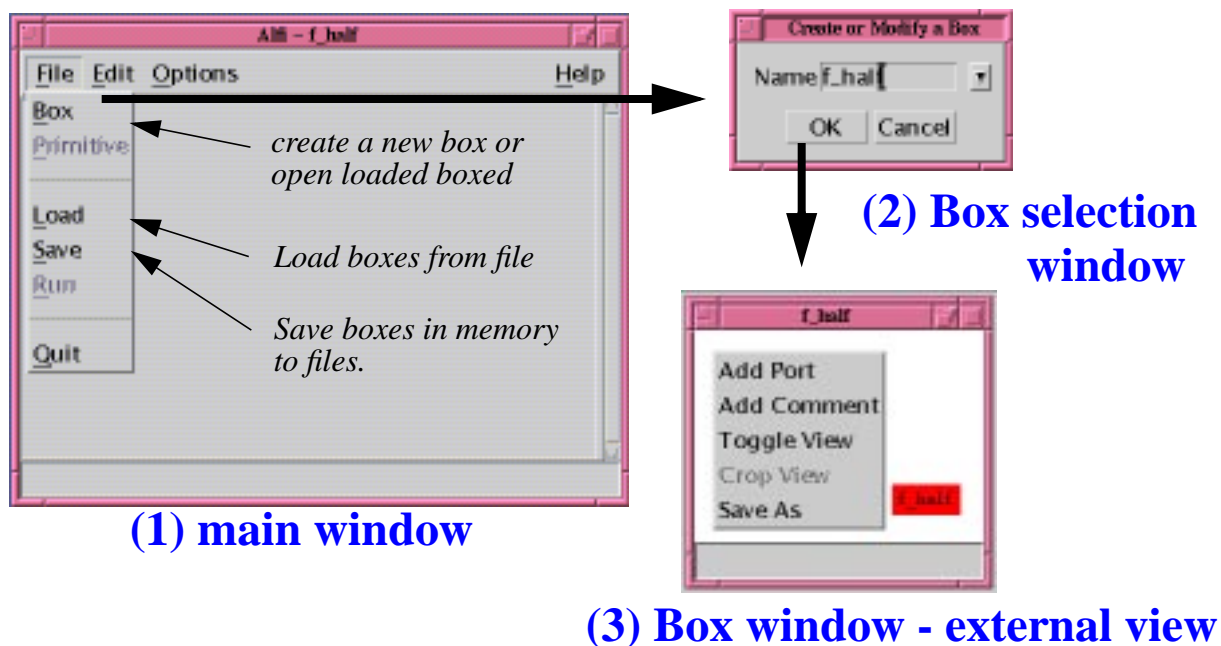
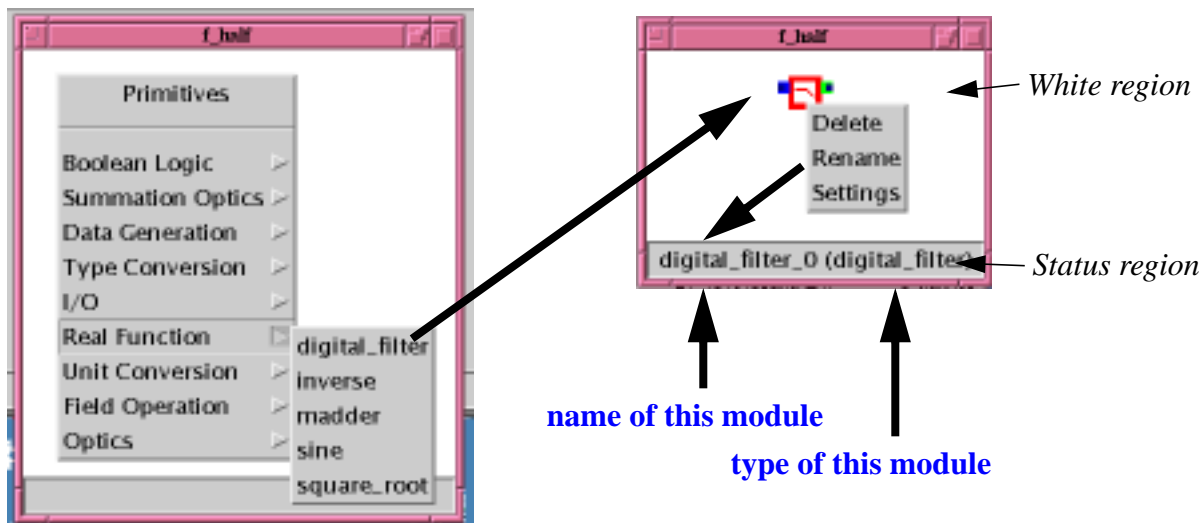


Figure 1: Starting up alfi

This box window is a view of the box from outside. Click in the white region in the window using the right mouse button. A menu shows up, as is shown in the external view in Figure 1. Select “Toggle View” from the menu. The window becomes empty. Now you are looking inside of the box which contains nothing now and where you are going to put pieces together, or write a program.



**Figure 2: Adding primitives in the internal view**

Click inside the empty window using left mouse button. A popup menu of primitive modules shows up, as is shown Figure 2. Available primitives are summarized in Appendix 2 Example of primitives. For details about these primitives, please consult document no. T990004. Primitives are grouped into different categories. Choose “digital\_filter” from the submenu at “Real function”. An icon is added to the window. Move the mouse on top of the icon. The narrow field at the bottom of the window is the status region, where information about the object of interest are shown. Now, two names are displayed in the zone. The first one is the name of this instance of the module, which is automatically assigned, and the type of the module. In analogy with C programming, you have written

```
digital_filter digital_filter_0;
```

just as you would write

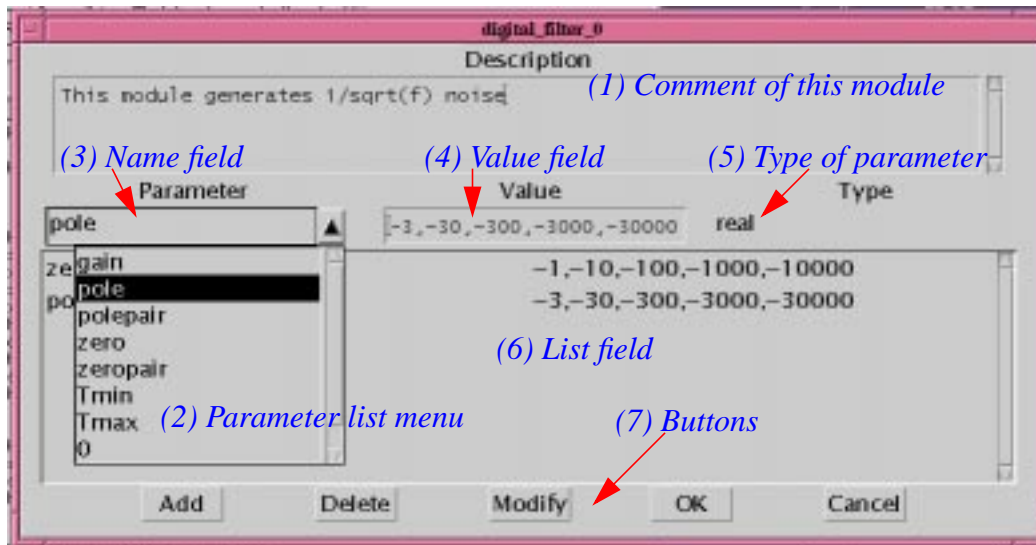
```
double ddd;
```

when writing a C source code. If you want to name the module yourself, click the icon using the right button. A menu pops up as is shown in the right window in Figure 2. Select “Rename” and you can change it. Just as a C function name does not directly affect the real performance, the naming is not crucial. But the name of the port, which corresponds to input and output parameters of C functions and will be explained soon, should be named in such a way that one can easily understand the meaning of the port. To remove the box, select “Delete”.

The digital\_filter in e2e is characterized by the following form, i.e., a overall normalization, real zeros ( $z_i$ ), complex zero pairs ( $z_{p_i}$ ), real poles ( $p_j$ ) and complex pole pairs ( $pp_j$ ).

$$G(s) = A \cdot \frac{\prod_{i1} (s - z_{i1}) \prod_{i2} (s - zp_{i2}) \cdot (s - \overline{zp_{i2}})}{\prod_{j1} (s - p_{j1}) \prod_{j2} (s - pp_{j2}) \cdot (s - \overline{pp_{j2}})} \quad (1)$$

Click the icon with the right button and you will see a popup menu in the right window in Figure 2. Select “Settings” from the menu, which brings up a window to define the settings of this module. Figure 3 is the settings window for the digital\_filter.



**Figure 3: Setting window**

At the top of the window is a comment on this module. You select the parameter to set from the parameter list menu. To generate the  $1/f^{1/2}$  spectrum, the method employed by VIRGO will be used, i.e., the filter is

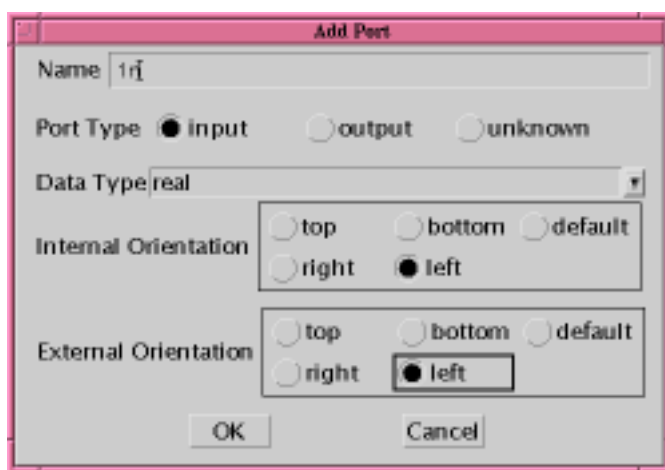
$$G(s) = \prod_i \frac{s - z_i}{s - p_i} \quad (2)$$

where  $z_i = (-3, -30, -300, -3000, -30000)$  and  $p_i = (-1, -10, -100, -1000, -10000)$ , where all zero and pole values are given in unit of rad/sec. To set a parameter, select the parameter, say pole, from the menu, and type in the value in the value field. When you select a parameter from the menu, the type of the parameter, e.g., real or integer, is displayed next the box where you enter the value. After you type in the value, you click “Add” button at the bottom. Then that setting appears in the List field at the bottom of the window. If you want to change the setting, click the parameter in the List field, change the value and click “Modify”. If you want to remove a setting, select the parameter in the List field and click “Remove” button.

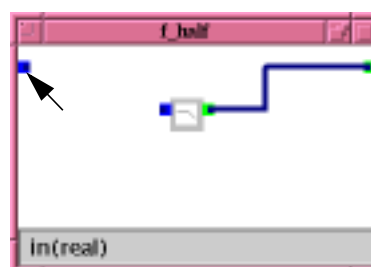
When all settings are done, **click “OK”**. **THEN SELECT SAVE FROM THE FILE MENU. DON'T FORGET TO SAVE.** All modifications are done in the memory, and when you select

“Save”, the content is saved in a file, whose name is the box name with “box” as the extension, e.g., the file name of the box “f\_half” is “f\_half.box”. The content in the file can be loaded into memory again using “Load” in the file menu. If you want to edit “h\_half” box later, choose “Load” from the menu, select “h\_half.box” file in the file selection dialog, select “Box” from the file menu, and choose “f\_half” from the pop-up menu in the window. When you load a box file, all related box files are automatically loaded and you don’t need to open each box file individually.

After closing the settings dialog, you will find the color of the box has changed from red to gray. This indicates that you have modified the box and customized the module.



(1) Port definition dialog



(2) Ports and connections

Figure 4: Adding ports to box

Next, we define the input and output for this box, through which this module communicates with other components. Click the white area in the window with the right button. A menu as shown in Figure 1 (3) appears. Select “Add Port”. The Port definition dialog appears as is shown in Figure 4. This dialog lets you define the input and output ports. Type the name of the port in the name field. Select “input”, make the data type to be “real”, and choose two “left”s and click OK. The blue box on the left edge of the window appears. The orientation buttons define where the port appears. Repeat the process to create another port, but this time, make it an output port on the right side of the box window.

**SELECT SAVE FROM THE FILE MENU, DO IT, DO IT NOW.**

As Figure 4 (2) shows, the digital\_filter box has its own ports, one on the left, one on the right. An input port is represented by blue and an output port by green. Connect the output port of the box to the output port of the window, by clicking one port followed by another click on the other port. Then a line is drawn as in Figure 4 (2), showing the data flow chain. Do the same for the input link. When you want to remove a link line, move the mouse pointer over the line, watching the status region to confirm that a proper link is under the pointer, click the right mouse button, and select delete from the popup menu, just as you do to remove modules.

Toggle back to the external view of the `f_half` box. The `f_half` box now has two ports. Move the pointer to one of the ports. The name and the data type of that port is displayed in the status region of the window.

What you have done corresponds to the following pseudo-C code.

```
void f_half( real in, real *out )
{
    digital_filter digital_filter_0;
    digital_filter_0( zeros = (-3,...), poles = (-1,...) );
    *out = digital_filter_0( in );
}
```

If you are interested in trying out how this filter works, go to Section 7.3. and follow the instructions.

### 6.3. Thermal noise module

You have learned basic techniques for using `alfi`. In this section, a module is constructed which generates thermal noise containing  $1/f^{1/2}$  tail plus one resonance peak, i.e.,

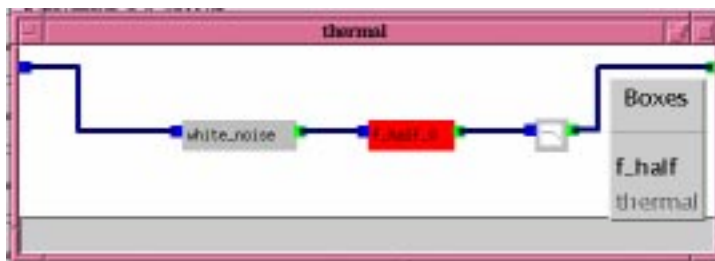
$$x(\omega)^2 = \frac{1}{\omega \left( (\omega^2 - \omega_0^2)^2 + \frac{\omega_0^4}{Q^2} \right)} \quad (3)$$

The resonant part is approximated by a double-pole low-pass filter with poles at  $(-\omega_0/2Q_0 \pm i\omega_0)$ .

Let us create a configuration box to generate a time series of data which has the distribution given in Eq.(3). Choose “*Box*” from the “*File*” menu, create a box named “`thermal`”, and switch to the internal view. Use the primitive menu and create a copy of module “`rnd_norm`”, which is in “Data Generation” submenu, and “`digital_filter`”. The “`rnd_norm`” module generates a random numbers with a normal distribution whose width is defined by the input “`width`”. This module is used to generate a white noise, so rename it to “`white_noise`”.

We need to use “`f_half`” box. Click in the white area in the “`thermal`” window using the left button with the control key down. You will see “Boxes” menu which has “`f_half`” in it. Select it to create an instance of “`f_half`”. This is like calling functions in C, but is more powerful - see the `alfi` manual. Create an input port of type real and name it “`noise_width`”, an output port of type real

and name it “out”. Then link all starting from the input port to `rnd_norm` to `f_half` to `digital_filter` module to the output port.



**Figure 5: Thermal box and “Boxes” menu**

The white noise goes through the `f_half` module to generate the  $1/f^{1/2}$  spectrum, then goes through the `digital_filter` to add the resonance structure.

We need to set some parameters. First for `white_noise` module. There is one parameter, “width”. This is the name of an input port of “`rnd_norm`” module, meaning that the value is being passed from other module. Set it to 1. What this does is to provide a default value if no input is connected (*Note: data coming inside through an input port overrides any setting value that you choose to put for the parameter of the input port*).

Next, open the `digital_filter` setting, and set “polepair” to (-0.5, 1000). You need to specify only one of the complex conjugate pair, and you use a syntax (real, imaginary) to represent a complex number, and you can enter more than one value by putting “,” between values.

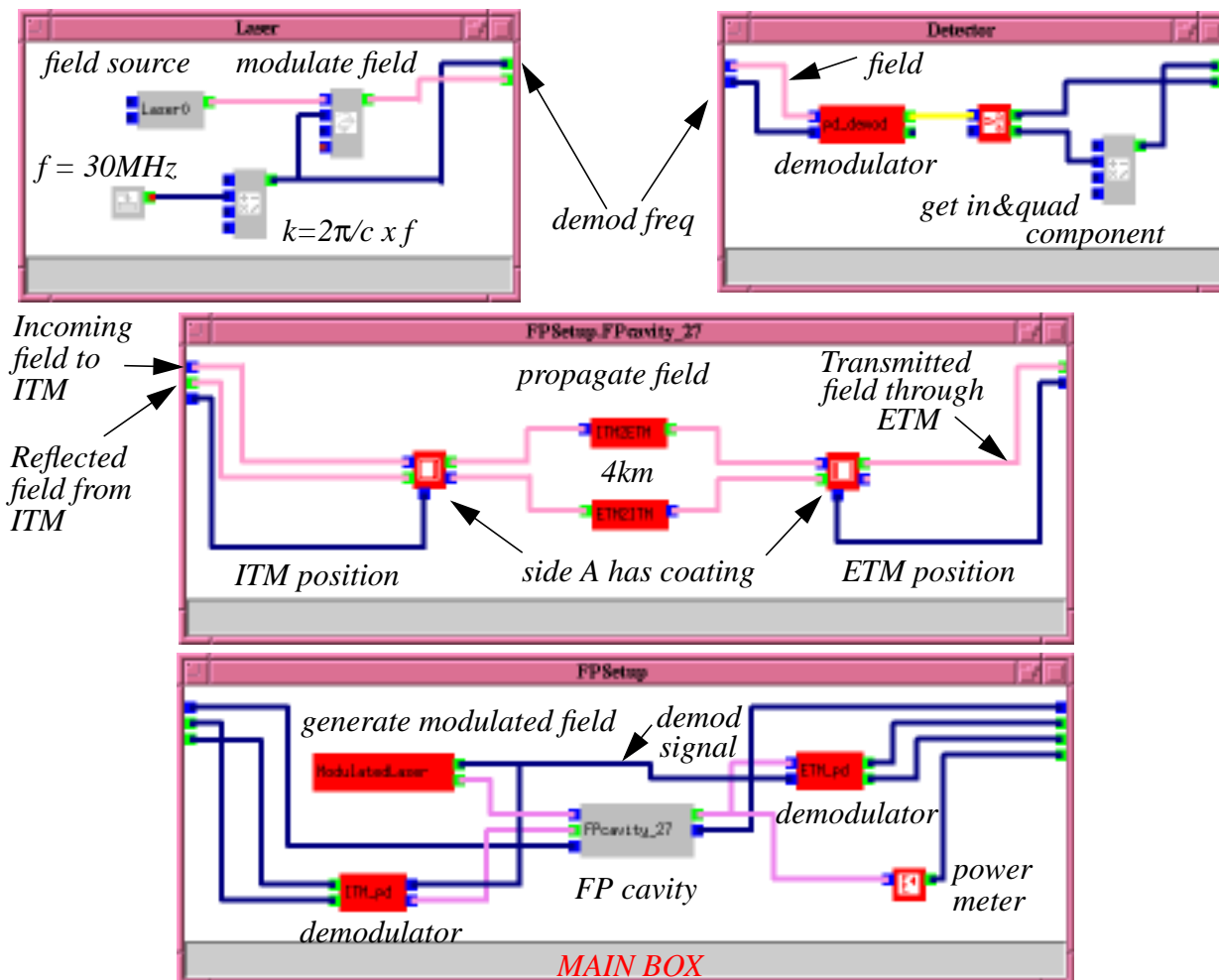
**SELECT SAVE FROM THE FILE MENU.**

## 6.4. Fabry-Perot example

In this tutorial, the following 4 box files are included: `Laser.box`, `Detector.box`, `FPcavity.box` and `FPSetup.box`. In this configuration, laser field is generated and modulated in `Laser.box`, passed into `FPcavity.box`, in which time evolution of the field is calculated with given values of mirror positions, and the reflected and transmitted fields are demodulated and detected. Consult document T9900004 to know more about the modules used in all these `.box` files.

One thing you have to be careful about is handling the optics, i.e., the side of the coating. By convention, the “`mirror2`” primitive has the coating on side A. In order to make the FP cavity, you have to link - let field propagates back and forth - sides A of both mirrors. For convenience, the module can be rotated or its sides can be swapped, so that side A, which usually faces to the left as ETM is now, can face to right. To rotate the module in whichever way you wish, move the mouse pointer on top of the module and select your favourite item from the pop-up menu: “rotate”, “swap”, “symmetry”.

In this example, we are interested in only longitudinal motion of mirrors. You may run this in a fast way by setting “`max_mode_order`” to ‘-1’ in “`field_gen`” (shown as `Laser0` in `Laser.box`) module which corresponds to plane-wave approximation (consult document T9900004). You do not need to set any setting related to transverse spatial dimensions (e.g., `waist_size_X` etc. in ‘`field_gen`’ and `radius_front` etc. in “`mirror2`” modules). Even if you put those inadvertently, Adlib would ignore those once you set `max_mode_order` = -1 as stated above.



**Figure 6: Fabry-Perot cavity configuration**

Using these box files, you can calculate transfer functions or the field response for a swinging mirror. Some results will be shown in the following section.

## 7 RUNNING E2E AND ANALYZING THE OUTPUT

### 7.1. What is this step for

In order to simulate the configuration which you have just created, you run programs provided as a part of the e2e package. The current version of these programs produces ascii file outputs of data of type real. The output is written each time new data are generated. The e2e package does not have data visualization software included yet. To see the data, use existing softwares like gnuplot or matlab or whatever you like which accepts the format explained in this section.

If you want to change the output format, you need to modify a software source code, modeler\_base and modeler\_freq. This is an advanced topic, and consult one of the core documents.

## 7.2. modeler and modeler\_freq

modeler and modeler\_freq are programs provided as a part of e2e package. The schematic structure is shown in Appendix 1 schematic view of e2e. The modeler reads in the main box file, perform the time domain simulation with a specified time step, and writes the time series of data coming out from the output ports of the main box file.

The modeler\_freq is a software spectrum analyzer: It does *sweep-sine* measurement in order to calculate transfer functions. It generates a signal of the form  $A \sin(2\pi f t)$  with a fixed frequency  $f$ , feeds that into one input port of a box, does the time domain simulation in exactly the same way as in modeler, waits until the output becomes stable, i.e., the output behaves as  $B \sin(2\pi f t + \phi) + C$  with  $B$ ,  $\phi$  and  $C$  being time independent, print out  $B/A$  and  $\phi$ , then changes the frequency to a new value and repeat the procedure.

In the following, we run these two programs, explaining how to interact with the program and how to analyze the output. In the following example runs, **blue** text corresponds to prompts from the program, **red bold** are the inputs you type. *Black italic* text between `<< and >>` are explanation for the inputs and outputs, so don't type. If you see `<return>`, a genius like you will be able to guess very easily that you need to press the return key.

## 7.3. $1/f^{1/2}$ filter

In the directory where you have the f\_half.box, type "modeler\_freq". The following summarizes the inputs to the program.

```
<< box file name >>
E2E> Description file >> f_half.box
<< just ., will be explained in the following example >>
E2E> Parameter File ('.' for none) (def="f_half.prm") >> .

<< frequency response is written here >>
E2E> Output file name ('.' to halt): (def="f_half.dat") >> <return>
E2E> This is a new file (def= yes) >> <return>

<< You can save all setting values in one file for later review >>
E2E> Do you want to save the settings ?
E2E> Setting File Name ('.' for no output): (def="f_half.set") >> <return>
E2E> This is a new file (def= yes) >> <return>

<< modeler_freq changes the time step depending on the frequency being analyzed.
Provide the widest possible safe range. There is no intrinsic time scale
in this spectrum, so it does not matter. >>
E2E> Set the range of acceptable time steps
E2E>
E2E>   longest_time_step  [0:INF]  = 1
E2E>   shortest_time_step [0:INF]  = 1
E2E>
```

E2E> "name = val", "?" for help or OK >> **l = 1, s = 1e-8**

<< you define which input port should be shaken and with what magnitude.

The asterisk \* in front of a port name indicates that it's selected. >>

E2E> Shaker input port name

E2E>

E2E> \*in

E2E> "name" ("+"/"-") to toggle on/off (all on/off) or OK >> **OK**

E2E> Amplitude of the sine signal (def=1e-14) >> **1**

<< you decide which output ports are to be analyzed. >>

E2E> Select output ports to be analyzed

E2E>

E2E> \*out

E2E> "name" ("+"/"-") to toggle on/off (all on/off) or OK >> **OK**

<< accuracy of result >>

E2E> convergence tolerance (def=0.01,[1e-05:1]) >> **<return>**

<< writes time series of "out" from the output port. This is useful

mostly for debugging purpose - you may like to look at this file if

calculation is taking too much time to converge, etc. However, remember,

in some runs, this may become very large as time progresses. >>

E2E> file to dump the time series ( "." for no dump ) (def="f\_half.dump") >> **<return>**

<< frequencies are changed from "start\_freq" to "end\_freq" with equal

spacing in log. "start\_freq" can be greater than "end\_freq" or vice versa>>

E2E> Set the simulation frequencies

E2E>

E2E> start\_freq [1e-05:70000] = 10000

E2E> end\_freq [1e-05:70000] = 10

E2E> num\_points [1:INF] = 10

E2E>

E2E> "name = val", "?" for help or OK >> **s = 10, e = 1e5, n = 100**

<< some information printed on screen for each frequency >>

frequency = 10.000000, time step = 9.765625e-04 = 9.765625e-03 / freq ( 1 of 20 )

Data is analyzed at every 1 th event and is dumped to file at every 5 th event

1.27051

frequency = 14.384499, time step = 4.882812e-04 = 7.023681e-03 / freq ( 2 of 20 )

Data is analyzed at every 1 th event and is dumped to file at every 7 th event

2.17822

....

....

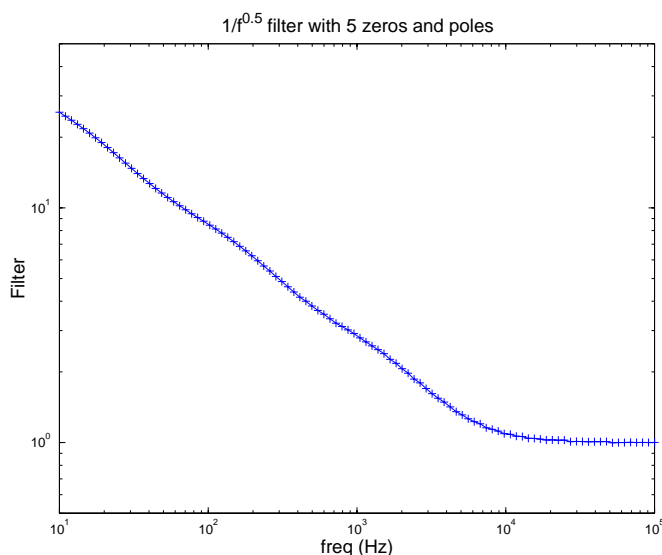
<< At the end you find this. If you want to continue more, say yes >>

```
E2E> Continue? (y or n) n
```

The output file `f_half.dat` is a text file which contains the following data (see Section 7.2. for the definition of  $B/A$  and  $\phi$ ).

input frequency	B/A	$\phi$	estimated freq	#time steps
1.0000000000e+01	2.5556754892e+01	-7.1993819177e-01	1.0000008455e+01	20669
1.0974987655e+01	2.4578555289e+01	-7.2354438064e-01	1.0974985260e+01	100194
1.2045035403e+01	2.3629155218e+01	-7.2852960511e-01	1.2045031697e+01	93334
....				

The first row explains the meaning of each column. The actual data file, however, has only numbers, no headers. The first column is the frequency of the source fed into the `f_half` “in” port, second is the magnitude of the transfer function, third is the phase. If there are more than one output port, this pair, magnitude and phase of the transfer function, of each output is repeated. The last two columns are performance measures. The column second from the last is the estimated frequency from the output, and this should be the same as the input frequency. The last column is the number of time steps needed to get the stabilized output. And the following is a plot with column 1 for the x-axis and column 2 for the y-axis.



**Figure 7: Transfer function of `f_half` box**

## 7.4. Thermal noise data

Type “`modeler`” to run the program to generate the time series of the output data. The following is the interaction with the program, using the same color convention as above.

```
<< the box file you want to simulate >>
E2E> Description file = thermal.box
```

```

<< thermal.box has one input, "noise_width". You can set the value here.
  You prepare a file which contains a line
      noise_width = 1.0
  and type the file name, say, thermal.prm which has this setting. >>
E2E> Parameter File ( '.' for none) (def="thermal.prm") >> <return>

<< output port data will be written in this file >>
E2E> Output file name ( '.' to halt): (def="thermal.dat") >> <return>
E2E> This is a new file. (def=yes) >> <return>

<< You can save all setting values in one file for later review >>
Do you want to save the settings ?
E2E> Setting File Name ( '.' for no output) (def="thermal.set") >> <return>
E2E> This is a new file. (def=yes) >> <return>

<< The time step of the simulation. If you need a resolution at frequency f,
  choose the time step < 0.1/f. >>
E2E> Model time step (def=1e-05,[0:INF]) >> <return>

<< how many seconds do you want to simulate. If you have chosen 1e-5 for the
  time step, and if you choose 10 seconds here, there will be 10^6 time
  evolution loop. >>
Current time is 0.000 (s).
E2E> Simulation time (s) (def=1,[0:INF]) >> 10

<< You may not need to store all the data points simulated.
  If you have chosen the time step to 10^-5, and if you choose N here to 100,
  then the data is stored to the output file at every 10^-3 seconds. >>
E2E> Write one data point every N steps. N (0 to halt) (def=1,[0:INF]) >> 100

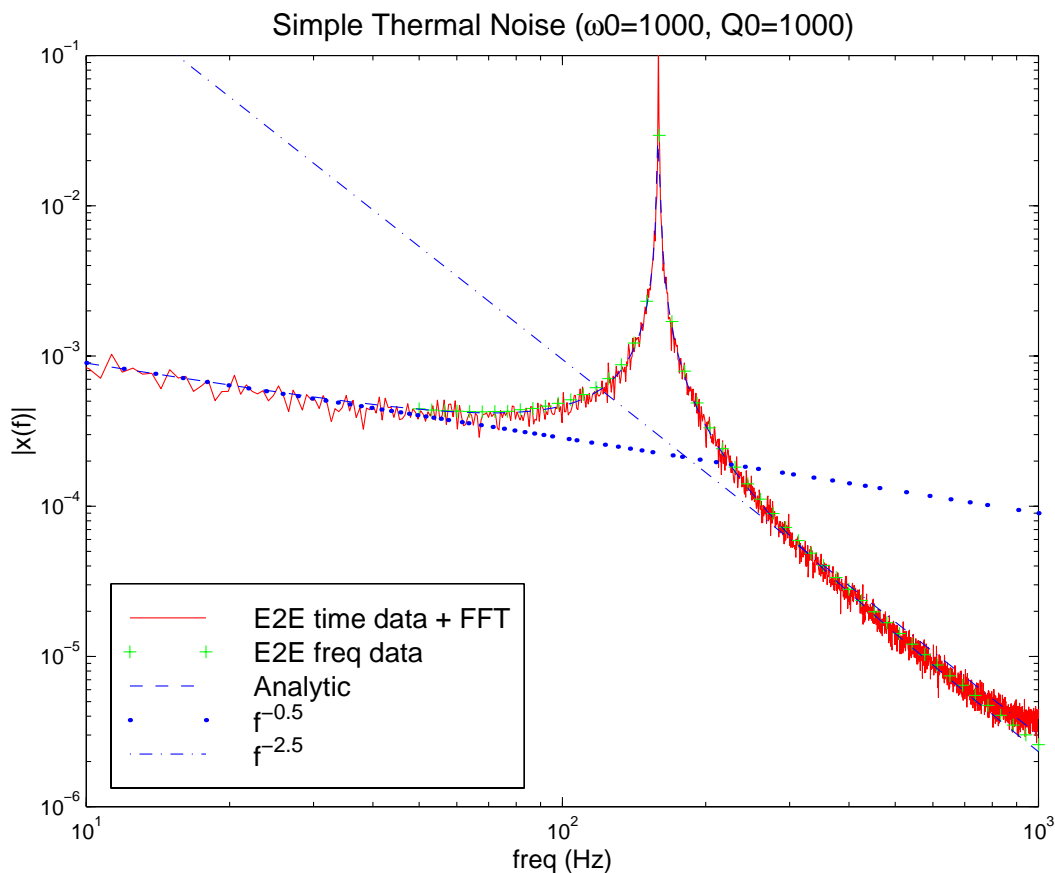
<<...and the modeler runs...>>
<< If you want to run more, you can say yes in the following. If you have
  changed the setting in thermal.prm, the simulation uses new values
  in this data file for the input of the simulation. >>
E2E> Continue? (y or n) n

```

The content of thermal.dat looks like the following.

time	value at output
0.0010	1.0034109326e-08
0.0020	1.5753470856e-08
0.0030	-4.7999670067e-07
0.0040	-1.0085298240e-06
...	

Essentially time and value at the output port. If there are more than one output, they will be repeated in the same row. The FFTed result using matlab is shown in Fig. 8. In the same figure, the result obtained running modeler\_freq and the analytic form Eq.(3) are also shown. Also



**Figure 8: Frequency spectrum of thermal noise**

superposed are two shapes,  $1/f^{1/2}$  and  $1/f^{3/2}$  for reference. As can be seen from this figure, the model reproduces the shape well.

## 7.5. Fabry-Perot - X func

Using the box files discussed in Sec. 6.4., we will calculate the transfer function and the response when the mirror swings. First the transfer function from the ETM to the reflected field. You run `modeler_freq` and specify the `FPSetup.box` as the description file. In the following, some parts of the input are explained, using the same coloring convention.

```
<< When using optics, the time step is important. You have to choose
an integer fraction of one-way-trip-time. modeler_freq adjusts the
time step depending on the audio modulation frequency. For the upper
limit, enter a number close to 0.01/f_max, where f_max is the maximum
frequency of interest. >>
```

```
E2E> Set the range of acceptable time steps
E2E>
E2E>   longest_time_step  [0:INF] = 1
E2E>   shortest_time_step [0:INF] = 1
E2E>
```

LIGO-DRAFT

```
E2E> "name = val", "?" for help or OK >> l = 0.6667e-5, s = 0.6667e-7
```

<< If the worst-case estimated error using the input time step exceeds 10%, the following warning is printed. This message could appear during the run-time when the frequency is changed and the time step has to be changed. If you see messages, check if the frequency range (the third line of the message) is satisfied. >>

```
-> Root.FPcavity_27.ITM2ETM * 200 internal delay steps.
-> Root.FPcavity_27.ITM2ETM * 12.8451% time step error.
-> Root.FPcavity_27.ITM2ETM * Use this for frequency << 1.1677e+06
```

<< There are several inputs, and you have to choose which one to shake.

The asterisk \* in front of a port name implies that it has been selected>>

```
E2E> Shaker input port name
```

```
E2E>
```

```
E2E>      *ETMx          ITMx
```

```
E2E>      Root.ModulatedLaser.freq
```

```
E2E>
```

```
E2E> Select 1 item
```

```
E2E> "name" ("+"/"-") to toggle on/off (all on/off) or OK >> OK
```

<< The amplitude should be chosen small enough to stay in the linear region >>

```
E2E> Amplitude of the sine signal (def=1e-14) >> 1e-13
```

<< There are 5 outputs, and you choose which one to be analyzed.

In the final output, a pair of data, magnitude and phase of the transfer function, for the selected output ports are placed.

To know in which order the outputs appear in your data file, see FPSSetup.dhr file automatically generated by modeler\_freq.

Switch off the Transpower port.>>

```
E2E> Select output ports to be analyzed
```

```
E2E>
```

```
E2E>      *Transpower      *InETM      *QuETM
```

```
E2E>      *InITM          *QuITM
```

```
E2E>
```

```
E2E> "name" ("+"/"-") to toggle on/off (all on/off) or OK >> Transpower
```

```
E2E>
```

```
E2E> Select output ports to be analyzed
```

```
E2E>
```

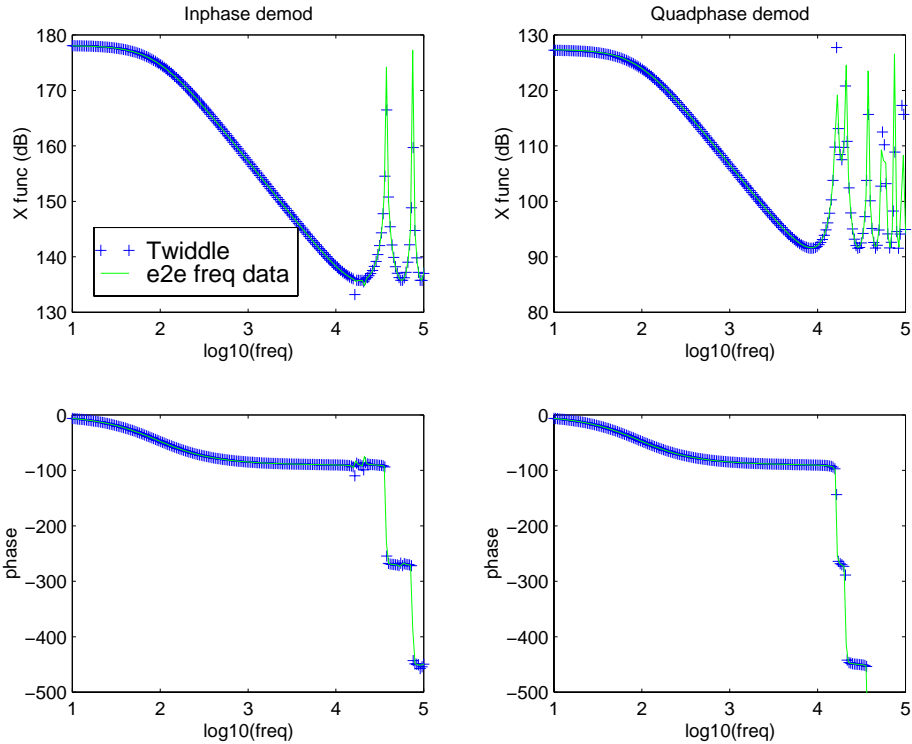
```
E2E>      Transpower      *InETM      *QuETM
```

```
E2E>      *InITM          *QuITM
```

```
E2E>
```

```
E2E> "name" ("+"/"-") to toggle on/off (all on/off) or OK >> OK
```

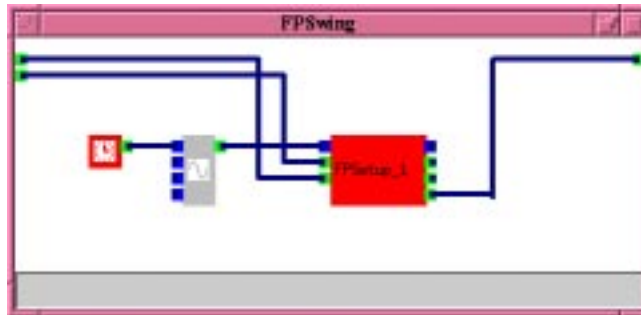
The transfer function from ETM motion to the reflected field is shown in Fig. 9, together with results obtained by frequency-domain calculations using Twiddle. Both magnitude and phase agree well.



**Figure 9: Transfer function from ETM to ITM reflected field**

### 7.6. Fabry-Perot - swinging mirror

In order to swing the mirror, we generate the z value (z representing small displacement of mirror

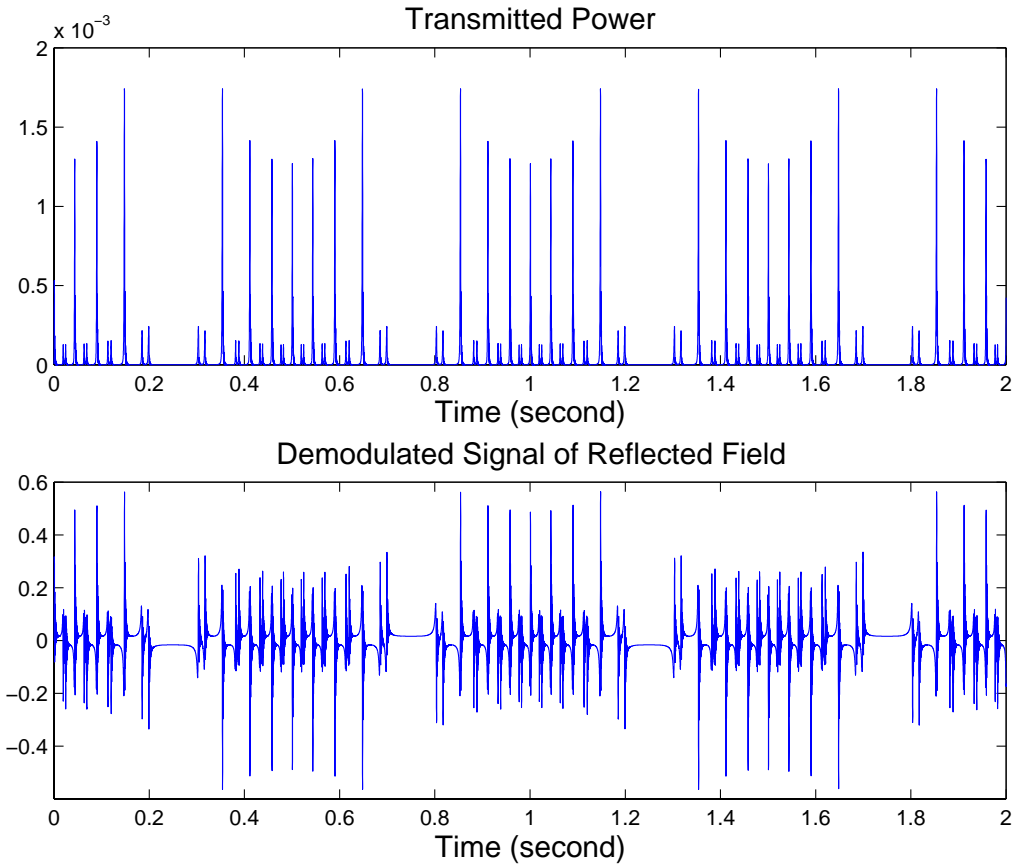


**Figure 10: mirror motion input**

in longitudinal axis which, by definition, is Z) and connect the value to the port ITMz of the box FPSetup, which is internally connected to the ITM mass position. The z-value is generated using a “sine” module, which generates the sinusoidal shape, with a “clock” module which provides the current time. The settings of the sine module is done so that it swings with an amplitude of 2 micrometer with a frequency of 1Hz.

LIGO DRAFT

Run modeler with FPSwing.box as the description file with a time step of  $0.6667e-5$  and you get results as shown in Fig. 11. This is consistent with the measurement of the 40m one arm open loop measurement.



**Figure 11: Swinging mirror**

## 8 ADDING A NEW FUNCTION

### 8.1. What is this step for

Adlib, the simulation engine of e2e included in modeler and modeler\_freq, has many functionalities in it, like “field\_gen” or “sideband\_gen”. We call these primitive modules. When some new functionality is needed, one can add the new capability. The C++ based modular design allows you to concentrate on the physics you want to add anew, not programming issues like the interaction with the time-loop. In this tutorial, the “madder” primitive, which represents just an algebraic manipulation, will be explained in detail. Another document, **Extending E2E Model - The How to Guide to Coding Modules** (LIGO-T980067) gives examples written in C++, C and FORTRAN.

## 8.2. simple example

The following is an excerpt from `math_function.h` and `math_function.cc`. This is a module to add a new function,  $a*x + b*y$ . If you need to add a new module which accepts several real value inputs and generates a new value, only thing you need to do is to modify the place shown in red-italic.

```
// class declaration. Use as it is, just simply substitute your class name
class madder : public real_function
{
    public:
        madder(const string& name_arg = "", const module* parent_arg = NULL);
        ~madder();
        module* new_type(const string& name_arg, const module* parent_arg) const;
        void action();
};

// code implementation
madder::madder
(const string& name_arg, const module* parent_arg)
// In the following, the third argument is the module name
// and the last argument is the number of inputs.
: real_function(name_arg, parent_arg, "madder", 4)
{
    // set_input_name(id,name) gives a name to the input port.
    // set_default_input(id,val) gives a default value when nothing is connected.
    set_input_name(0, "a");
    set_default_input(0, 1.0);
    set_input_name(1, "x");
    set_default_input(1, 0.0);
    set_input_name(2, "b");
    set_default_input(2, 1.0);
    set_input_name(3, "y");
    set_default_input(3, 0.0);
}

// just that
madder::~~madder()
{}

// just copy as it is, but use your module class name
module* madder::new_type
(const string& name_arg, const module* parent_arg) const
```

LIGO-DRAFT

```
{ return new madder(name_arg, parent_arg); }

// This is where you spend most of your intelligence.
void madder::action()
// in(0) = a, in(1) = x, in(2) = b, in(3) = y
{ output = in(0) * in(1) + in(2) * in(3); }
```

In short, to implement a new module which looks like

```
double func( double a1, double a2, ... )
```

you need to just specify the number of inputs, their names and default values, and, of course, the content of the function.

### 8.3. Real story

If you want to do more, you need to do more, but the program structure is designed to minimize the overhead. Once the e2e environment is setup, you need to change 2 to 3 lines in Makefile, possibly in modeler\_base.cc, and type make. No fine prints anywhere else.

You can create a new application which can generate different kind of outputs. The modeler\_base is a base class of the application framework, which is explained in the “Overview” document. You can create a derived class of modeler\_base, and you can concentrate on the change you want to make, without bothering how to make a time loop or read in the description file.

## 9 FREQUENTLY ASKED QUESTIONS

### 9.1. How to use a beam-splitter?

Use a combination of two “mirror2” to represent a beam-splitter. We supply such a ready-made BS.box file which has four inputs and four outputs.

### 9.2. What is the order of data in the output file?

When an output file named xxx.dat is created, another file named xxx.dhr (xxx matches to the data file name, not literally xxx) is automatically created. This file contains names of the outputs, one name per line, in the order they are placed in the data file. E.g., if the xxx.dhr contains the following lines, the first column in the xxx.dat file is time, second column comes from data\_out module named amp in box CR\_00 in box FP.

```
time
FP.box.CR_00.amp
FP.box.FF_0_InDemod
```

### 9.3. How can I define the order of the output?

When the program creates an output file named xxx.dat, it looks for a file named xxx.dhr. If there is a file named xxx.dhr, it uses the order in that file to arrange the order of the data whose names match with the names in the given xxx.dhr file. E.g., the content of the existing xxx.dhr is as follows.

```
time
FP.box.CR_00.amp
FP.box.SB_00.amp
FP.box.FF_0_InDemod
FP.box.FF_0_QuDemod
```

And the names of your data are

```
time
FP.box.FF_0_QuDemod
FP.box.FF_0_InDemod
FP.box.SB_00.amp
FP.box.CR_00.amp
FP.box.SB_10.amp
FP.box.CR_10.amp
```

Then, the order of the columns in the data file is

```
time
FP.box.CR_00.amp
FP.box.SB_00.amp
FP.box.FF_0_InDemod
FP.box.FF_0_QuDemod
FP.box.SB_10.amp
FP.box.CR_10.amp
```

The order of the first 5 data are determined by the original xxx.dhr, and the rest of the data are placed in the order they appear in box files involved in the simulation run, which is hard to predict. When a new data file is created, the original xxx.dhr file is updated to reflect the the new order. One can change only the order of data coming from data\_out, i.e., you cannot change the placement of time or frequency.

### 9.4. How can I save my key strokes when I run modeler or modeler\_freq, so that I don't need to retype again ?

When you start running modeler or modeler\_freq, there are three special commands for that purpose.

@(filename) : open a file and start saving key strokes in that file.

@) : stop recording key strokes. If you reached the end, you don't need to worry.

@filename : play back the key strokes stored in the file.

Once stored, you can use it also as the source of the pipe input to modeler / modeler\_freq as  
 modeler < filename

## 9.5. How can I use this feature in my program?

Use functions implemented in `e2ecli.cc` and `e2ecli.h`. Five top level functions are

```
double e2ecli_getDbl( "prompt", "help", default_val, min_val, max_val );
```

```
int e2ecli_getInt( "prompt", "help", default_val, min_val, max_val );
```

```
bool e2ecli_getBool( "prompt", "help"); no default value
```

```
bool e2ecli_getBool( "prompt", "help", default_val );
```

```
void e2ecli_getStr( "prompt", "help", &str ), str may have the default value on entry, on  
return it has the new value.
```

**modval** and **inquire** are functions to let the user change related values together.

When the user types "?" mark, the "help" text is displayed, and when the user simply types "return" key, the default value is returned if a default value is given.

## 9.6. How can I implement a frequency noise?

All frequencies of subfields, the carrier and sidebands, are constant during the simulation, and they cannot be fluctuated. The frequency noise should be implemented as a phase noise in the following way:

$$\phi(t) = \int_0^t \omega(t) dt = \omega_0 \cdot t + \int_0^t \delta\omega(t) dt \quad (4)$$

The first term is the constant frequency part, and the second part is the noise. In stead of changing the frequency, the phase of the subfield is incremented by this amount.

# 10 TROUBLE-SHOOTING

**Trouble:** *You edited a setting in your box file, ran modeler, waited, plotted data file, but got confused by looking at it. The data is much different from what you expect from your settings.*

**Shoot:** Did you forget to save your .box file after you made your changes?

**Trouble:** *You created a cavity.box file, had set all its parameters and so, as you expected, all modules were looking grey. Then you included that inside another box file called test.box. You clicked on cavity.box which immediately opened up. What the hell is going on here?!@! All modules inside cavity.box are red!*

**Shoot:** By clicking on cavity.box you've opened up the local copy of the original cavity.box sitting on its own in your directory. We call the copy as internal box (so, instead of clicking on the icon of cavity.box, if you pop-up the menu on cavity.box and select "view internal" you would still get the same box with all red-colored modules inside that). You can view the original box (with all grey settings) by selecting "view basebox" from the same menu. Settings in the local copy or the internal box are all red because this shows only the differences it has from the base box. If you make a change locally, then the corresponding module in internal box will turn grey.

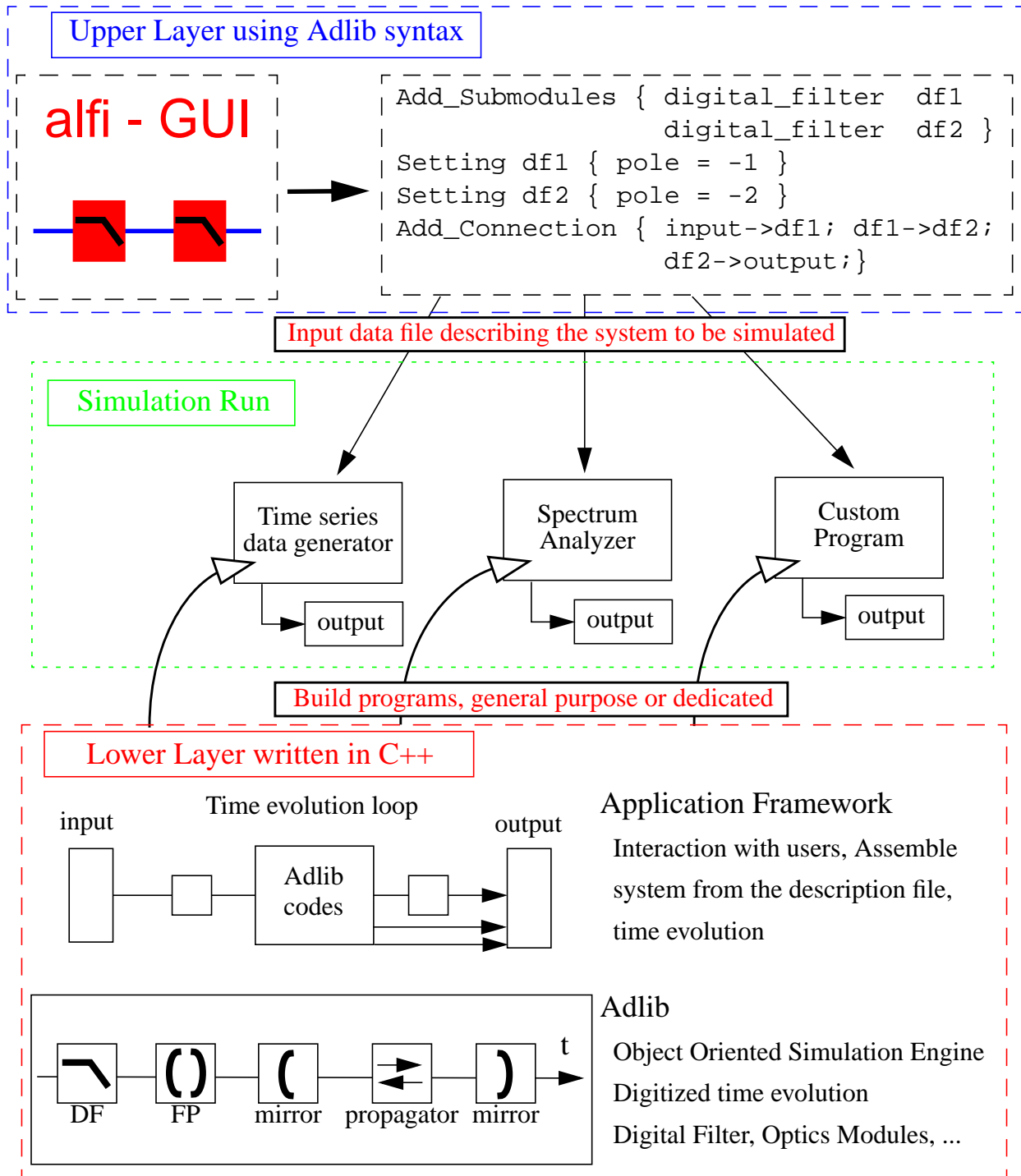
**Caution:** Be careful when you decide to change the base box. It'll change all local copies in all boxes that contain it.

## 11 WE CARE FOR YOU

Please give us feed back what are missing. We really wish to make this and other documents useful.

LIGO-DRAFT

# APPENDIX 1 SCHEMATIC VIEW OF E2E



## APPENDIX 2 EXAMPLE OF PRIMITIVES

The following is a part of primitives built in e2e. The latest version of the full list of primitives and explanations are available in T990004\_multimode.fm5 (Last updated : 011/1/99).

<i>Name</i>	<i>Function</i>	<i>setting</i>
<b>Field Operation</b>		
field_gen	generates a field using Hermite Gauss mode expansion	wave length, power, beam waist data, polarization, single mode or multi mode, phase and amplitude noise
sideband_gen	modulates phase and amplitude of a field	modulation depth, frequency, number of side bands, noise
pd_demod	photo diode and demodulator.	demodulation frequency, detector shape, shot noise
<b>Optics</b>		
mirror2	a 2-input 2-output mirror	mirror specifications ( reflectance, transmittance, curvature, reflective index ), incident angle of input field, displacement and misalignment.
propagator	propagates a field over a macroscopic distance	length of the propagation
telescope	Simulate a collection of lenses	specification of lenses, output beam profile
<b>Summation Optics (short cavities)</b>		
cav_sum, tricav_sum, rec_sum	represents FP, isosceles triangular, recycled Michelson cavities	static length between mirrors, mirror specifications, displacement and misalignment of mirror.
<b>Real Function</b>		
digital_filter	a digital filter	zeros and poles
A2D_sampler	Analog to Digital converter	time step
limiter	models a circuit with rails	upper and lower limits
<b>Data types</b>		
field	amplitudes of sidebands and modes, sideband freq., mode base, polarization	
clamp	position, rotation, force and torque. Generic data type for mechanical data.	
generic	integer, real, complex. vectors, string, boolean	

## APPENDIX 3 VARIOUS KIND OF FILES

There are several files used in e2e. They are discriminated by the extension in the file name, and are summarized in the following table. In the table, Adlib means applications built using Adlib library and an application framework, like modeler and modeler\_freq. All files are ascii text files.

**Table 2: File kinds**

<i>.ext</i>	<i>input or output</i>	<i>what</i>
.box	output of alfi input to Adlib	This file contains the description of the system to be simulated using e2e. You use alfi to create and modify this file. When you run an Adlib program, you choose a box file to specify what is to be simulated. With this file, you can specify input ports (see below about .par file) and output ports specifying data to be generated and stored in the output file.
.par	output of text editor input to Adlib	In a box file, you can attach input ports or data_in modules where you need to provide values. You can specify constant values of these inputs in this .par file. E.g., a=1 will set values to 1 of all input ports or data_in modules whose names are a. If you specify B.a = 1, then data_in module named a in a box file B are given this value.
.set	output of Adlib	This file contains all the settings used in this simulation run.
.dat	output of Adlib	Run results of Adlib programs are saved in the .dat file. The first column is time when a time series is generated, and if frequency when a transfer function is calculated.
.dhr	output of Adlib input to Adlib	Together with each .dat file, one .dhr file is created. This file contains the title of the data in the data file in the order they are placed in the data file. E.g., if a .dhr's content is time - power - amp ( three lines, one name per line) The first column in the data file is time and the second and third columns are data associated with output ports named "power" and "amp". When an Adlib program creates a new .dat file, it looks for an associated .dhr file, and if there is one, the order specified in the .dhr file is used in the .dat file. If you want to place "amp" before "power" in the example above, you need modify the .dhr file to time - amp - power ( three lines, one name per line)
.prm	out put of alfi input to alfi	The specification of each primitive, like number of inputs, is stored in this file. When you edit box files, alfi reads these files in the E2E_PATH, and you can choose them from a menu.
.xbm	output of bitmap input to alfi	The icon of a primitive is stored in this file. Each .prm file contains the name of the .xbm file of the primitive.