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

Document Type	**LIGO--T000047-00 -**	**E**	Apr. 2000

e2e primitive module
- Reference Manual -

Biplab Bhawal, Matt Evans,
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/

1 WHAT IS THIS DOCUMENT

This document contains the complete and most up-to-date list of primitive modules of the End to End LIGO simulation program. The physics implemented in each module is described briefly in this document, and the details of the physics and formulations are given in separate documents ([1], [2], [3], [4]). The common process is to use alfi, a GUI front end of e2e, to combine these primitives to define a configuration to be simulated using the physics simulation program, save the configuration in a file (.box is an extension of the file name), and the simulation program reads this file when it runs. In Ch.6, the syntax of this description file is provided in case you need to deal with the content of the file directly.

2 USING THE PROGRAM - STEP BY STEP

2.1. A quick overview for E2E-user:

For using the end to end simulation programme, it is not necessary to know about the structure of source codes. However, knowledge of a few basic features may turn out to be useful. The following discussion assumes that you have already gone through our other document “Getting Started with E2E”).

The End-to-End (popularly called E2E) simulation codes have been written with the object-oriented approach of C++ language. The code is modular. Each component is almost independent of others.

In order to set up your own experiment, the first step is to properly place your individual instruments and components. E2E provides these: e.g., field_gen (alias laser source), sideband_gen or phase_adder (alias phase-modulator), pd_demod (the detector), mirror2 (2 inputs and 2 outputs) or mirror4 (4 inputs and 4 outputs), lens, power_meter etc. You need to do this job of assembling by creating what we call ***.box** file using our graphical interface, Alfi, or writing your description file (see document “Getting Started with E2E”). The next obvious step is to connect all these components meaningfully together and bring them to life. In an optical experiment, this is done by laser. However, we intellectuals, prefer to call it “field”.

Our field is a class which, at its heart, contains important information about laser light in the form of a vector of a vector: Each element of the parent vector represents a frequency of light (carrier or sideband), whereas each element of the offspring vector represents the complex coefficient of the amplitude of laser in a particular mode of Hermite-Gaussian basis. The basis of these modes is also carried by the field class itself in the form of its two important private members: waist-size of beam and distance to waist. As will be explained in sec. 2.1 below, this class also carries some important information about how you wish to perform your experiments.

The basic task of each module is to accept some input field and/or data and provide some output field and/or data. These can interact with each other directly or with the help of another important module, **“prop”**, the propagator (if these are exchanging fields and there is a distance between them).

We also developed some modules which represent composite representations of some primitive modules, e.g., “cav_sum”, a Fabry-Perot cavity or “rec_sum”, a recycled Michelson cavity. Of course, one can form a FP cavity or Michelson cavity using primitive modules of mirrors and props. However, inside these composite modules which are just like black-boxes, calculations of many round-trips are performed with the help of ready-made formulas and thus, if we need, we may use them for fast computation.

In next two subsections we describe all modules, their inputs, outputs, other parameters and also various data types that these modules use.

2.2. Data types and existing modules

Table 1: "Data types" summarizes data types used in the multi-mode version of Adlib, defining settings for modules and passing data between modules. “type name” is the name used for the documentation purpose, 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 “field” also use adlib_real for the real variable. The default is double type.

Table 1: Data types

type name	*description*	*example*	*data type*
complex		zeros and poles of digital filter	adlib_complex
vector_complex			array1d<adlib_complex>
integer		number of sidebands of field	int
vector_integer			array1d<int>
real		reflectance of mirrors	adlib_real
vector_real		power or phase of field_gen	array1d<adlib_real>
field		input and output of optics objects	field
string		type specification of data_in	string
boolean		freq_flag of power_meter	bool
clamp	data representing position, rotation, force and torque. Explicit form is defined in adlib_types.h. Nth bit of clamp.flag is true if Nth data is meaningful, i.e., if (flag&(1<<N) != 0) meaningful.	mirror position and rotation, connection between mechanical modules.	clamp
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 3: "Primitive Modules" is a table of all primitive modules. The details of modules are given later. The units of quantities used in these modules are as follows.

Table 2: Units

Quantity	*Unit*
length	m
time	second
Power	watts
Frequency	either Hz or rad/sec. (see module description)
Field	$\sqrt{watts}$
angle	radian
$k = 2\pi/\lambda$	m^{-1}
boolean (in setting)	yes/no or true/false
boolean (logic unit)	real value is used to represent true or false status. A value represent true if it is larger than “threshold”, false otherwise.

For many modules, the main input and output are named as “0”. When appropriate, the meaning is placed in () following the “0”.

Table 3: 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
data_viewer (Sec. 4.11.)	Interavively view data	"0" variable type	none	none
Real Function				
madder	implements z = a*x + b*y	"a"(1.0) "x"(0.0) "b"(1.0) "y"(0.0) real	"0" real	none
sine	the sine function out = amplitude x sin(2 π t + ϕ)	"0" (time) "amplitude" "frequency" "phase" real	"0" real	none
square_root	the square root function out = sqrt(in)	"0" real	"0" real	none

Name	*Function*	*in*	*out*	*setting*
inverse	the inverse function out = 1 / in	"0" real	"0" real	none
digital_filter (Sec. 4.14.)	a digital filter out = digiatl filter (in)	"0" real	"0" real	"zero" "pole" (in rad/sec.) "gain" real "zeropair" "polepair" complex “DenominatorPoly” (1) (real vector) “NumeratorPoly” (1) (real vector) “needPrecision” integer (none) “sampelTime”(0) real
A2D_sampler (Sec. 4.14.)	Discritize the input with the specified sampleTIme.	“0” real	“0” real	“sampleTime”(0), “integrationTime”(0) real
VAL_digitizer	digitize the input value using finite number of bits.	“0” real	“0” real	“min_val”(0), “max_val”(1) real “num_bits”(1) integer
limiter	models a circuit with rails if in < lower, out = lower if in > upper, out = upper	"0" "upper" "lower" real	"0" real	none
delay	add one delay explicily.	“0” 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.**				
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 the input value “bool” is true, the input value “high” is returned as the output, else the input value “low” is returned.	"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 out = 2 π / in	"0" real	"0" real	none

Name	*Function*	*in*	*out*	*setting*
f2k	converts frequency to wavenumber out = 2 π in / c	"0" real	"0" real	none
Type Conversion				
field2complex (Sec. 4.9.)	converts a field to a complex number	"0" field, "dk" real, “m”, “n” integer	"0" complex	none
field2info	gives info about the field	“0” field	“spot_size” real	none
complex2reim	converts a complex number to real and imaginary real = Re(in * exp(i phi)) imag = Im(in * exp(i phi))	"0" complex, "phi" real	"real" "imag" real	none
complex2aphi	converts a complex number to amplitude and phase amp = abs(in * exp(i phi)) phi = Arg(in * exp(i phi))	"0" complex	"amp" "phi" real	none
clamp2xyz	convert clamp to indiviual components and flag	“0” clamp	“X”,’Y”,”Z”,”thetaX”, “thetaY”, “thetaZ”, “FX”,... real “flag” integer	none
xyz2clamp	Combine individual data to make a clamp data. flag is automatically calculated based on the link.	“X”,...,”thetaX”,... real	“0” clamp	none
real2vec	Convert a real value to a vector of real with one data out = in,just type changes	“0” real	“0” vector_real	none
Field Operation				
field_gen (Sec. 4.1.)	generates a field	"power" vector_real, "phase" real (0.0)	"0" field	"lambda" real (1.064e-6), “waist_size_X”, waist_size_Y real(0.01), “distance_waist_X”, “distance_waist_Y” real(0.0), “max_mode_order” integer(1), “polarization” integer(1), “compute_option” integer(1) , “angle_resolution” real(1e-8) , “compute_mismatch_curvature” bool(no)
sideband_gen (Sec. 4.12.)	phase and amplitude modulates a field (uses sideband approximation)	"0" field, "k_mod", "gamma", “gammaAmp” real	"0" field	“order” integer
sideband_filter	passes only sidebands with dk value less than or equal to dk_max	"0" field, "dk_max" real	"0" field	

Name	*Function*	*in*	*out*	*setting*
fld_modulator (Sec. 4.16.)	modulate phase&amplitude of a field directly out = in * (1+del_amp) * exp(i*phi)	"0" field, "phi" , “del_amp” real	"0" field	none
freq_shifter (Sec. 4.15.)	shift frequencies of all subfields by del_k	“0” field “del_k” real	“0” field	none
power_meter (Sec. 4.2.)	outputs the power of a field	"0" field, “dk_for_power” real,	"0" real	“freq_flag” boolean; “meter_flag” , “m”, “n”, “order_min”, “order_max” integer ,
pd_demod (Sec. 4.13.)	photo diode with shot noise and demodulator.	"0" field, "k_demod" real	"demod" complex, "power" real	“shape” integer (0), “shotnoise” integer (0), “efficiency” (1.0)
Optics				
prop (Sec. 4.3.)	propagates a field over a macroscopic distance	"0" field	"0" field	"length" real (1.0) "dphi" real (0.0) “have_delay” bool (yes)
mirror2 (Sec. 4.4.)	a 2-input 2-output mirror (cavity end mirror)	"mech_data” clamp; "Ain" "Bin" field	"Aout" "Bout" field	"r" "t" "R" "T" "L" real (2.0), "angle"real(0.0), “radius_front”, “radius_back” real (1e20) , “refractive_index(1.0) real
telescope (Sec. 4.10.)	Simulate a collection of lenses	“in” field “length” real	“out” field	“waist_X”, “waist_Y”, “dist2waist_X”, “dist2waist_Y”,“guoy00_X”, “guoy00_Y” real “lensInfo” vector_complex (real part keeps the location and the imaginary part keeps the focal length of one mirror). “thicknessInfo” vector_real (thickness of each lens) “calc_sb_phase” bool (true)
Summation Optics:				

Name	*Function*	*in*	*out*	*setting*
cav_sum [b] (Sec. 4.6.)	represents a FP cavity	“mech_dataA”, “mech_dataB”, clamp; "Ain" "Bin" field,	"Aout" "Bout" "Apick" 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), "rB" "tB" "RB" "TB" "LB" real (2.0), "rC" "tC" "RC" "TC" "LC" real (2.0), “refractive_indexA”, “refractive_indexB” real (1.0), “radius_frontA”, “radius_frontB”,real(1e15), “radius_backA”, “radius_backB”, real(1e15).
tricav_sum (sec.2.9)	represents an isosceles triangular cavity	“mech_dataA”, “mech_dataB”, “mech_dataC” clamp; “Ain” field,	“Aout”, “Bout”, “Cout” field	“length_large” real(1.0), “length_small” real (0.01), “dphiAB”, “dphiBC”, “dphiCA” real (0.0); “rA” “tA” “RA” “TA” “LA” real (2.0), “rB” “tB” “RB” “TB” “LB” real (2.0), “rC” “tC” “RC” “TC” “LC” real (2.0), “radius_frontC” , real(1e15), “refractive_indexA”, “refractive_indexB”, “refractive_indexC” real (1.0),
rec_sum [c] (Sec. 4.7.)	represents a recycled MIFO	“mech_dataA”, “mech_dataB”, “mech_dataC”, “mech_dataD” clamp; "Ain" "Bin" "Cin" "Din" field	"Aout" "Bout" "Cout" "Dout" "Bpick" "Cpick" "Dpick" 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), "rB" "tB" "RB" "TB" "LB" real (2.0), "rC" "tC" "RC" "TC" "LC" real (2.0), "rD" "tD" "RD" "TD" "LD" real (2.0), “refractive_indexA”, “refractive_indexB”, “refractive_indexC”, “refractive_indexD” real(1.0), “radius_frontA”, “radius_frontB”, “radius_frontC”, “radius_frontD”, real(1e15), “radius_backA”, “radius_backB”, “radius_backC”, “radius_backD”, real(1e15).

3 CONVENTION

The curvature of a optics surface is positive (negative) if the surface looks concave (convex) from outside the optics element. Focal length is positive (negative) for converging (diverging) lenses.

Throughout this document X and Y represent horizontal, and vertical axis respectively. Z is the direction of beam-propagation in an unperturbed state of the optical set-up. The mechanical data (longitudinal position z, transverse shifts dx and dy, pitch and yaw) are attributed to a mirror (mirror2 or any mirror in a summation cavity) through a port called “**mech_data**” whose data_type is “**clamp**”. The following subsection describes the module which should be used to put mechanical data to mirror(s).

3.1. “xyz2clamp” module:

Inputs of this module (available under item “type_converters” in the pop-up menu of Alfi) are z, x, y, theta_x(pitch), theta_y(yaw) and whose output can be connected to “mech_data” port(s) of optics modules.

“**z**” : small longitudinal displacement of mirror. The sign is positive if the displacement is in the direction of normal to the coated surface.

“pitch” or **“yaw”** : “pitch” is rotation around the horizontal axis and “yaw” is rotation about the vertical axis. Consider the normal to the front (coated) surface of a perfectly aligned mirror. Let us call it z-axis. Now you know the positive x-axis and y-axis in a right-handed frame (consider x to be the horizontal axis). If the mirror rotates such that its normal now has a positive y-component then the “pitch” value is to be set positive. Similarly, If the mirror rotates such that its normal now has a positive x-component then the “yaw” value is to be set positive.

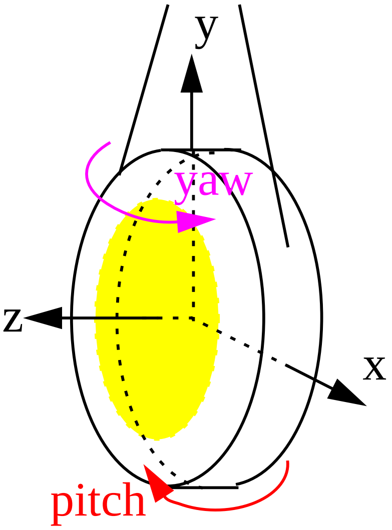


Figure 1: Defnion of axis and angle

3.2. Definition of length between optics

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

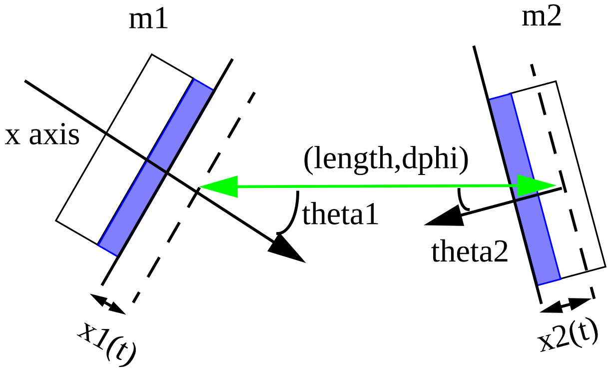


Figure 2: Definition of length

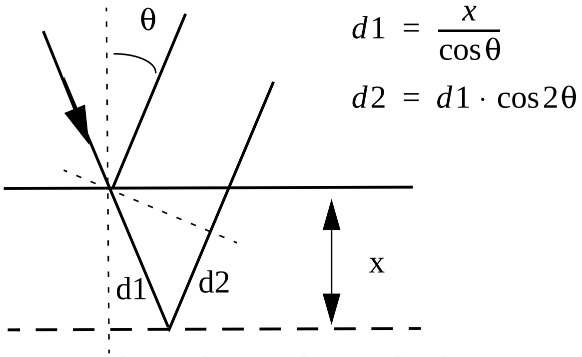
In Figure 2, the distance between the two mirrors, m1 and m2, are calculated using four quantities, length, dphi, x1(t) and x2(t). For each mirror, there is a reference place, shown using dashed lines in Figure 2, which is time independent. The distance between two reference planes is given by $L_0 = \left(N\left[\frac{length - lguoy00}{\lambda}\right] + \frac{dphi}{2\pi}\right) \cdot \lambda$, 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 deviation 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 2) 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

---

[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.

taken into account by the change of the phase. As is shown in Figure 3, the net change of the path



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

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

Figure 3: 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 SOME PRIMITIVE MODULES IN OPTICS:

In summation modules (**cav_sum, rec_sum, tricav_sum**), there are some restrictions which should be noted carefully. We decided to keep these restrictions in order to avoid unnecessary options which are not really utilised in LIGO-related applications that we know of. It should be noted that any or all of these restrictions can be lifted by a quick modification of our source programme; In case you need such modifications, please contact us.

4.1. field_gen:

This is basically our laser source but it also carries some important additional information about how you wish your simulation to be done. Optical simulation without light means nothing. A mirror or a cavity is alive only when it receives light. That's why we decided to put these additional information inside this module. The field carries these additional information (or the user-specified instructions) everywhere it goes and simulation is performed accordingly everywhere in a consistent way. So we explain below the parameters of this module in two catagories:

4.1.1. simulation information:

"max_mode_order": represents the maximum order (m+n of TEM) upto which the user wishes to perform the computation. As explained above, once specified, this remains to be a static constant throughout the simulation. If you set "max_mode_order = -1" or any other negative integer, all modules perform operations assuming light as plane wave (no tranverse dimensions). Setting "max_mode_order" to zero or other positive integer (upto 3) makes all the modules perform Gaussian beam calculations using multi-mode computational environment; The zero setting corresponds to just TEM00 mode. Note that the current implementation can study modes upto order m+n = 3, which is sufficient for most of our application purposes.

“compute_option” : allows the user to select one of the computational methods for the multi-mode calculations. Currently, only one option , 1, the standard modal-model computation, is available. NOTE: if you set “max_mode_order” to any negative integer, which effectively means that you wish to perform ordinary single-mode operations, obviously, the setting of “compute_option” will not have any significance and will be ignored.

“angle_resolution” : Matrices that are used to study higher order modes generated due to pitch and yaw are updated only if these quantities (in radian) get changed by at least the set-value of this parameter. Thus, this avoids expensive matrix re-calculations even for negligible changes in alignment angles. Choice of a higher value leads to relatively less (not necessarily unacceptable) accuracy but faster simulation, and vice versa.

“compute_mismatch_curvature”: This is a boolean flag. If you wish to compute for the generation of higher order spatial modes due to mismatch in radii of curvature of mirrors and the corresponding phase-fronts, you need to set it to either true or yes. If you set it to false or no, the simulation assumes that the phase-front at any mirror exactly matches with the radius of curvature of the corresponding mirror. This has many advantages. For example, when you are at the first stage of designing some configuration, you may not be interested in detailed mismatch calculations. Caution: before setting it to no or false, be sure that mismatches are really small.

4.1.2. field information:

“lambda”: laser wave-length.

“polarization”: At present E2E supports field in only one polarization state and does not allow their simultaneous presence (This status will be changed shortly). Set this parameter to either “0” (zero) if the field has p-polarization (in the plane of incidence - XZ plane in E2E’s convention) or to “1” if the field has s-polarization (perpendicular to the plane of incidence - YZ plane).

“waist_size_X”, “waist_size_Y” : laser beam waist radii : Radial distance in X or Y direction at which the electric field drops to 1/e times the maximum value (at the center).

“distance_waist_X”, “distance_waist_Y” : Distance in z-direction to beam’s waist: To be set negative (positive) for a converging (diverging) beam.

“power” and **“phase”**: These in various modes need to be specified as an array of real numbers in the following order of TEM_xy basis: 00, 10, 01, 20, 11, 02, 30, 21, 12, 03. Note that the current implementation can study modes upto order m+n = 3, which is sufficient for most of our application purposes. If it is really necessary, we’ll incorporate m+n > 3 modes in future.

Some examples: if you set max_mode_order = -1 or 0 (single-mode simulation) and power = 1.0, 0.2, 0.1, only TEM00 power will be set to 1.0; the last two values in the array are ignored. If you set max_mode_order = 1 and power = 1.0, 0.2, 0.1, 0.01, the last value in the array is ignored. If you set max_mode_order = 1 and power = 1.0, 0.2, the TEM01 power is automatically set to zero.

4.2. power_meter:

“dk_for_power”: Difference between the frequency for which you intend to measure power and the carrier frequency. If you set it to zero, that means you intend to measure carrier power. NOTE: you must set **“freq_flag”** to yes in order to use this parameter.

“freq_flag”: if you set it to “yes”, the “power_meter” module calculates power in frequency corresponding to the set value of **“dk_for_power”** . If you set the same to “no”, it sums up power in all frequencies. In both cases, it sums up power in only those modes selected by you by setting “meter_flag”.

“meter_flag” : Setting “meter_flag” to zero, you get summed-up power in all modes. If it is set to 1, power_meter sums up power in all modes in between m+n = **“order_min”** to m+n = **“order_max”**; The settings of **“m”** and **“n”**, if you make any, will be neglected. When “meter_flag” is set to 2, the power_meter gives the power only in mode TEM_mn; In this case, the settings of **“order_min”** or **“order_max”**, if any, are neglected. If you are doing something inconsistent (e.g., “order_min” is greater than “order_max”, etc.), you’ll receive warning messages right at the start of your run of modeler or modeler_freq. So, watch out for those and, if needed, stop running and change the settings.

An easy question: How to get total power in all frequencies and in all modes? Answer: Set “freq_flag” to no and “meter_flag” to 0.

4.3. prop (the propagator):

“length” and **“dphi”** : The total length of a propagation path is calculated as follows:

$$L_0 = \left(N\left[\frac{length}{\lambda}\right] + \frac{dphi}{2\pi}\right) \cdot \lambda \tag{1}$$

In the equation, N[x] means the closest integer to x, and λ is the carrier wavelength. When dphi = 0, the propagation path length is an integer times the wave length.

“have_delay” : When “have_delay” is true, prop behaves as a module with delay, i.e., at least one time step delay is introduced, even if the length is 0. So, maximum time-step of simulation is determined by maximum value of “length” parameters of all the props involved. However, When “have_delay” is false, prop calculates the output by multiplying proper phases without any time delay. This is intended to simulate a very short cavity and field paths outside of a resonator. Use of this latter modus-operandi may speed up the simulation speed without introducing any extra inaccuracy.

4.4. mirror2:

Side A (B) referes to the side which is coated (uncoated). E.g., A_{in} means an input field coming into the coated side.

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

“radius_front”, “radius_back”: Radius of curvature of the coated surface. To be set positive (negative) if the coated surface looks concave (convex) from outside the mirror.

“refractive_index”: refractive index of substrate

“angle” : The angle between the incident or reflected beam and the normal to the mirror surface. When “angle” = 0, the mode-matching between the input beams and the mirror surface is assumed; Any small difference between “radius_front” and the radius of wavefront of the beam is then computed in a perturbative way (provided you keep **“compute_mismatch_curvature”** to yes or true in **“field_gen”**). However, when “angle” is not zero, the mirror is treated as a turning one. Incoming and reflected beams are related by ABCD transformation which uses the value assigned to “radius_front”. Effects of mirror rotation (pitch, yaw) are calculated in a perturbative way.

“mech_data”: see section 3 "Convention"

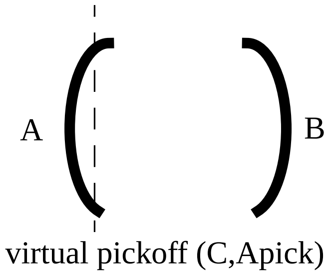
4.5. lens:

Module removed. Use telescope instead.

This module may be used to effect the change of basis of beam TEM modes by a lens or by a mirror with lensing action. To use it for studying the lensing effect of a mirror, please refer to the first paragraph of section 2.4 on mirror2.

“radius_front” and **“radius_back” : To** be set positive (negative) if the lens surface looks concave (convex) from outside the lens. “radius_front” is on the side of “in” field and “radius_back” is on the side of “out” field.

4.6. cav_sum:



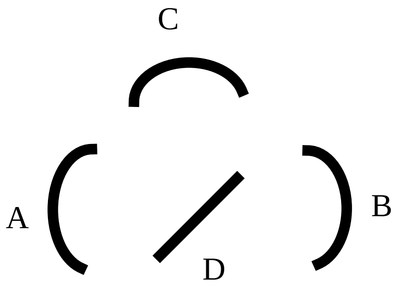
This is used for fast simulation of a Fabry-Perot cavity. *There is one restriction in this module:* The first light should enter the cavity through mirror A.

The coated sides of mirrors, by default, are inside the cavity. In case you need to orient one or both of them otherwise, set **“dirA”** and/or **“dirB”** to (-1).

Lensing effects of the component mirrors have been included in calculations. So, do not forget to set **“refractive_index”**, **“radius_front”** and **“radius_back”** of mirrors A and B.

Give mechanical data of the mirrors through “**mech_dataA**” and “**mech_dataB**” ports (see section 3 "Convention").

4.7. rec_sum:



This represents the recycling cavity of LIGO interferometer or just a power-recycled Michelson interferometer. *There is one restriction in this module:* The first light should enter the cavity through mirror A.

This has been developed in order to perform fast simulation of the whole LIGO interferometer. In a LIGO configuration made with primitive mirrors and propagators, the maximum value of time-step of simulation is limited by the smallest value of one of the lengths (in this case, one of the lengths inside the recycling cavity). This module enables one to make a LIGO configuration where "rec_cav" sits in the middle and gets joined by the props to the primitive end mirrors and allows a time-step whose maximum value is limited by the lengths of arm cavities. Of course, it can, on its own, produce simulation results for a Michelson interferometer in a fast way. It can also be used to study dual-recycled michelson interferometer by having non-delay props and primitive signal recycling mirror at its dark port.

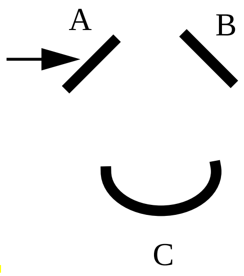
By default, the coated sides of all the mirrors are inside the power-recycled Michelson Cavity. To simulate with one or more than one coated sides turned to outside this configuration, set corresponding "dir_" variable to (-1). For example, in order to study a power-recycled Michelson cavity, most probably what you would like to simulate is just the default orientation of mirrors in "rec_sum". However, if you wish to study full LIGO configuration using "rec_sum" for the recycling cavity, you need to set **dirB** and **dirC** to (-1).

Lensing effects of the component mirrors have been included in calculations. So, donot forget to set **"refractive_index"**, **"radius_front"** and **"radius_back"** of each mirror.

Give mechanical data of the mirrors through "**mech_dataA**", "**mech_dataB**", ... ports (see section 3 "Convention"). Remember: the longitudinal position, z, of the beam-splitter refers to shift along the normal to its coated surface, just like in any other mirror (and the $\sqrt{2}$ factor is taken care of by the code).

The output fields **Apick**, **Bpick**, **Cpick** refer to internal fields at corresponding mirrors and are directed at the beam-splitter. The field **Dpick** is the field at the beam-splitter and is directed to mirror B.

4.8. tricav_sum (isosceles triangular cavity):



This is a summation module representing a triangular cavity like pre-mode-cleaner or mode-cleaner. *Four restrictions on this module*: (i) the triangle should be an isosceles one, (ii) light should enter only one port (referred to as A port), (iii) the input (A) and output (B) mirrors should be flat., (iv) the coated sides of all mirrors are always inside the cavity.

"length_large": Either of lengths BC or CA.

"length_small": length AB**.**

"radius_frontC" : radius of curvature of mirror C**.**

"refractive_indexA", "refractive_indexB", "refractive_indexC" : refractive indices of mirrors

“dphiAB”, “dphiBC”, “dphiCA”: small phase offsets in various lengths.

If all dphi_ are zero, the triangular cavity would be resonant with TEM00 of its natural modal basis in p-polarization. So, if you have set “polarization” to “0” in field-gen module of your .box file and if all dphi_ are zero, the cavity will automatically be resonant. However, you need to set one of the dphi_ s to Pi to make it resonant if you have set “polarization” to “1” (i.e. s-polarization) in field-gen module.

Give mechanical data of the mirrors through “**mech_dataA**”, “**mech_dataB**”, ... ports (see section 3 "Convention").

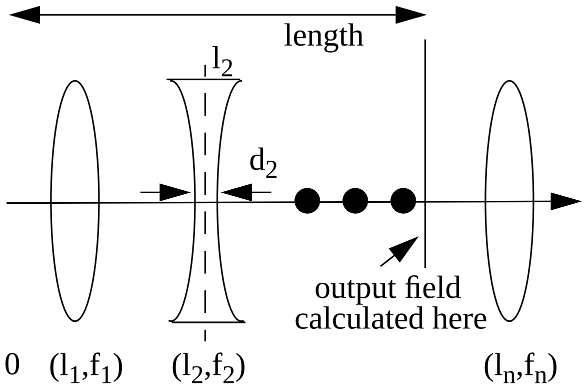
4.9. field2complex:

This module allows one to get the complex amplitude of a field (which, by E2E construction, is a class containing various field information and associated functions) in frequency specified by **“dk”** (as usual, the difference between the specified frequency and the carrier frequency) and in a particular TEM_mn mode specified by integers **“m”** and **“n”**.

4.10. telescope

Telescope module simulates a set of thin lenses to change the waist size and position, and the phase of the field. The lens setting is defined by its location l_i and the focal length f_i, optionally with its thickness d_i, where the focal length is related to the lens surace curvatures, R_1 and R_2, and its refractive index n_{ref}, by the following equation.

$$\frac{1}{f} = -(n_{ref} - 1)\left(\frac{1}{R_1} + \frac{1}{R_2}\right)$$



The “**lensInfo**” setup should be defined in the following way to define the lens configuration.

$$lensInfo = (l_1, f_1), (l_2, f_2), \ldots, (l_n, f_n)$$

If you want to include the thickness effect, you provide “**thicknessInfo**” in the following format

$$thicknessInfo = d_1, d_2, \ldots, d_m$$

When there is a thicknesses assigned, the lens position l_i is the center between two surfaces. If the thickness information is not specified for the j’th lens, zero thickness is assumed. The thickness is used only to correct for the calculation of the waist position, and no thick lens effect is included.

In order to use this module to simulate one lens, set “**lensInfo**” to (l,f), where “l” is the distance between the source of the field and the lens, and “f” is the focal length.

The “**length**” of the telescope can be defined through the input port, and it can vary during the simulation. If “**length**” is not provided neither as an input to this port, nor by a default value, the last lens location is used as the length of the telescope. If neither of them are provided, the length is set to be zero. The output of the telescope module is the field at the location “**length**”.

The field is propagated between lenses in the same way as the propagator module does, i.e., guoy phases and sideband phases ($(l_m - l_{m-1})*dk_i$) are applied and the distance to the waist position is advanced accordingly. When the field goes through a lens, the waist size and the distance to the waust position is changed. If the focal length is larger than 10^{10}, it represents a flat lens.

When the sideband phases are included, the definition of the demodulation of the field after the telescope, in-phase and quad-phase, depends on the length of the telescope. In order to make it easy to define the in-phase and quad-phase demodulation, the sideband phases can be excluded from the telescope calculation. In order to do that, set “**calc_sb_phase**” to false.

The telescope effect can be specified by the setting parameters “**waist_X**”, “**waist_Y**”, “**dist2waist_X**”, “**dist2waist_Y**”, “**guoy00_X**”, “**guoy00_Y**” in stead of specifying the details of the lens setting. If these parameters are specified, the base of the outgoing field is changed to these values and, each mode is multiplied by a phase based on guoy00. In this case, no sideband fields are multiplied.

If “**lensInfo**” is specified and one or more of these three parameters are specified, these parameter settings override the calculation based on the lensInfo specification, i.e., after the calculation of the telescope is finished using the “**lensInfo**” data, the final waist size, the distance to waist and total gouy phase changes are replaced by the explicit specification, if there were any.

4.11. Data_Viewer

This is a module to dump out the data. This is equivalent to the following c++ statement.

```
for ( i = 0; i < counter*step; i++ )
  if ( mod(i,step) == 0 )
    cout << data;
```

You are prompted for the values of counter and step, and you can stop dumping if you want. This data will not go to the standard output file.

4.12. sideband_gen (this one is not quite up-to-date)

This modules amplitude and phase modulates the input field by the following formula.

$$\begin{aligned} E_{out} &= E_{in} \cdot Exp(i\Gamma_{\varphi}\sin(\Omega t)) \cdot Exp(\Gamma_{amp}\sin(\Omega t)) \\ &= E_{in} \cdot \sum_{N=-\infty}^{\infty}\left(\sum_{i=-\infty}^{\infty}(-i)^{N-i} \cdot J_i(\Gamma_{\varphi}) \cdot I_{N-i}(\Gamma_{amp})\right) \cdot Exp(iN\Omega t) \end{aligned} \tag{2}$$

where $J_n(x)$ is the Bessel function and $I_n(x)$ is the modified Bessel function. For the given value of “**order**” setting parameter n, the following approximation is used:

$$E_{out} \approx E_{in} \cdot \sum_{N=-n}^{n}\left(\sum_{i=-n}^{n}(-i)^{N-i} \cdot J_i(\Gamma_{\varphi}) \cdot I_{N-i}(\Gamma_{amp})\right) \cdot Exp(iN\Omega t) \tag{3}$$

4.13. pd_demod

The details of the implementation of the demodulation and shotnoise are given in [1]. Setting “**efficiency**” is the quantum efficiency, which is multiplied to the input power to get the net power converted to the photo current.

There are three options for the shot noise simulation. When “**shotnoise**” is 0, the shot noise is not simulated. When “**shotnoise**” is 1, a fast method is used to generate the shot noise. This generates the shot noise using a gaussian distribution which gives correct values for the average and the variance, when only one pair of sidebands (one upper and one lower) exists. This method generates the shot noise of the three signals, the inphase demodulated, quadphase demodulated and the power, independently. When “**shotnoise**” is 2, a full simulation is used to generate, and the simulated fluctuation is no more a simple poission distribution and the correlations among the three signals are properly generated. But this method is order of magnitude slower than the fast method.

The “**shape**” setting defines the shape of the detector. For the “shape” values 0 to 8, no additional inputs are needed, and each value corresponds to the shape shown in Figure 4 with infinite radius.

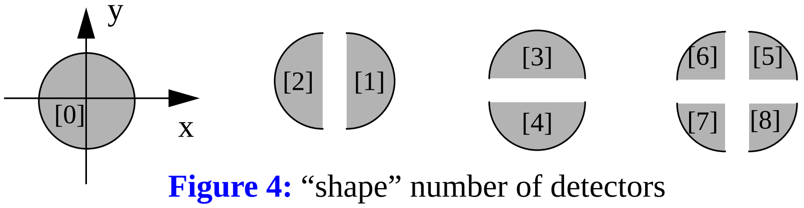


Figure 4: “shape” number of detectors

Several box files are provided, “**circular_det.box**”, “**xhalf_det.box**”, “**yhalf_det.box**” and “**quad_det.box**”. They contains one to four pd_demod modules with proper weights to combine them. complex2reim is included to convert the demodulated output to inphase and quadphase demodulated signals. In Figure 5, “+” and “-” signs indicate that they are added together with weights of 1 and -1 respectively.

When you need to simulate any detectors with different shapes, a detecotor map needs to be generated using a program “detmap”. [Contact Hiro Yamamoto of LIGO Lab about the details of this program] This program generates a table of values to be used by pd_demod for this detector. Then paste this table of numbers, array of real values, into the map_data field of pd_demod.

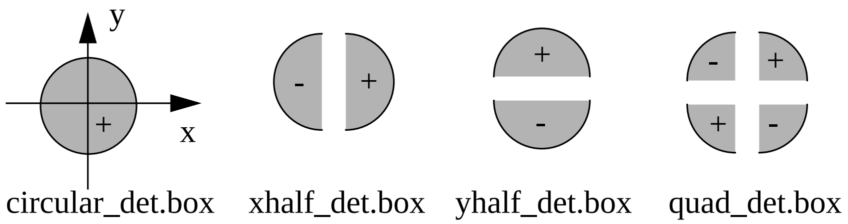


Figure 5: detector boxes

Using “detmap”, you can define a detector by specifying the following quantities (see Figure 6).

- r_min, r_max : minimum and maximal radius
- phi_begin, phi_end : minimum and maximal angle
- gap : distance between the detector boundary to the geometrical bound defined by phi_begin adn phi_end
- dx0, dy0 : the offset of the detector center to the beam center

All quantities with length dimention are to be normalized by the spot size.

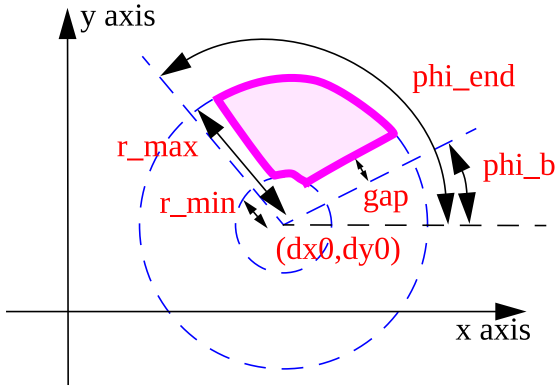


Figure 6: Specification of a detector

For example, if you want to define a Bullseye photodiode designed for IOO, you make detector maps of the following 4 detectors with the parameter sets (r_min,r_max,phi_min,phi_max) = (0,1,0,360), (1.15,2.748, -30,90), (1.15,2.748, 90,210). (1.15,2.748,210,330). The radius values are arbitrary chosen.

4.14. digital_filter

The digital filter module uses the Tustin or bilinear method, and is represented by the following form.

$$DF(s) = gain \cdot \frac{f_1(\vec{z}) \cdot f_2(\overrightarrow{zp}) \cdot f_3(\overrightarrow{zpo})}{f_1(\vec{p}) \cdot f_2(\overrightarrow{pp}) \cdot f_3(\overrightarrow{ppo})} \tag{4}$$

$$\begin{aligned} f_1(\vec{x}) &= \prod (s - x_i) \\ f_2(\overrightarrow{xp}) &= \prod (s - xp_i) \cdot (s - \overline{xp_i}) \\ f_3(\overrightarrow{xpo}) &= \sum xpo_i \cdot s^i \end{aligned} \tag{5}$$

The numerator and the denominator are represented by three forms of polynomials. zeros and poles in the form f1 in Eq.(5), complex zero pairs and poles pairs in the form f2 and polynomials in the form of f3. Each one is specified by a real vector (zeros, poles and polynomials) or by a complex vector. The coeeficients a's and b's are calculated for a given time step using 128 bit precision. When the new output value is calculated internally in the module, either 64 bit or 128 bit precisions are used depending on the values of zeros, poles and the time step. This criteria is not perfect. If you prefer to use 128 bit calculation for a given module, set "**needPrecision**" to 1. If you are sure that 64 bit is enough, set "**needPrecision**" to 0. If "**needPrecision**" is not specified, it is decided automatically.

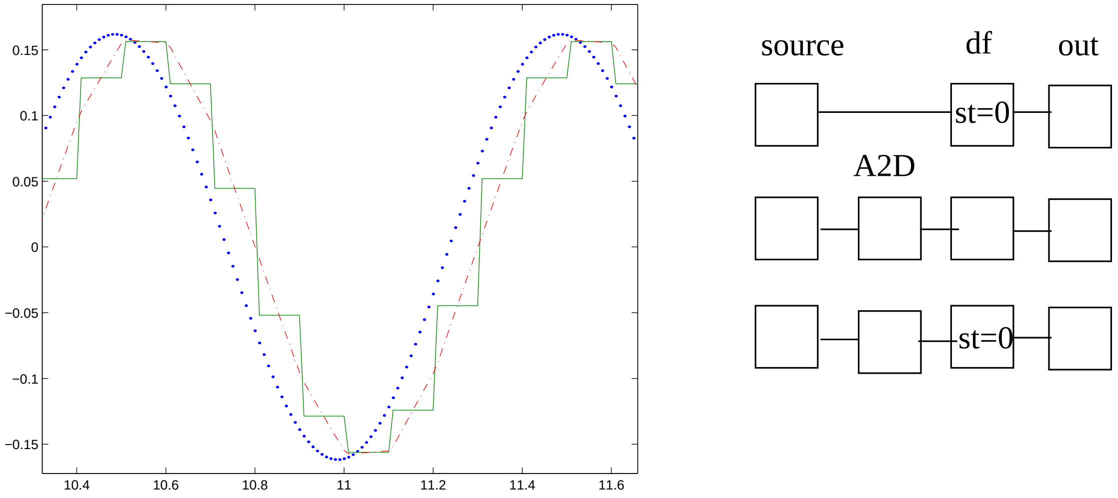


Figure 7: Digital Filter

"**sampleTime**" larger than the simulation time step (tick time), this module uses this value as the digitization time step. In Figure 7, the dotted line is the output with "sampleTime" = 0. The solid line is produced by placing a A2D_sampler between the source and the digital filter module, which has the same finite value of sampleTime as the digitial filter. The dot-dashed line is the the output of the same arrangement, source -> A2D_sampler -> digital_filter, but the sampleTime of the digital filter is set to 0.

4.15. freq_shifter

All subfield frequencies are shifted by the same amount. The magnitude of this shift can be several 100 MHz, it should not be time dependent.

4.16. fld_modulator

One can do the modulation using this function and demodulate by multiplying a sine function without using the sideband approximation. But, in order to do that, the time step should be at least 10 times smaller than the modulation field cycle, and usually, this method takes several 10-100 times slower than the side-band approximation. It is recomended that one tries this method occationally to validate something. When you set the number of sidebands for the sideband_gen, this is automatically done both in sideband_gen and pd_demod.

4.17. A2D_sampler

For a given discritization time period “sampleTime” τ and an integration time Δ, the output between nτ to (n+1)τ is calculated as

$$out = \frac{1}{\Delta}\int_{n\tau-\Delta}^{n\tau} input(t)dt \qquad (6)$$

When Δ is 0, the inpuit value at time nτ is used as the output value during nt to (n+1)t. When digital controllers are implemented, this module should be used together with the digital filter with the same “sampleTime”. There is no restriction of the sampleTime, except that it should be larger than the simulation time step. The numerical integration is done by using parabola fits at the two boundaries (area S1 and S2 in Figure 8) using 3 nearest data (three up arrows for S1 and 3 down arrows for S2) and by using extended Simpson’s rule in between.

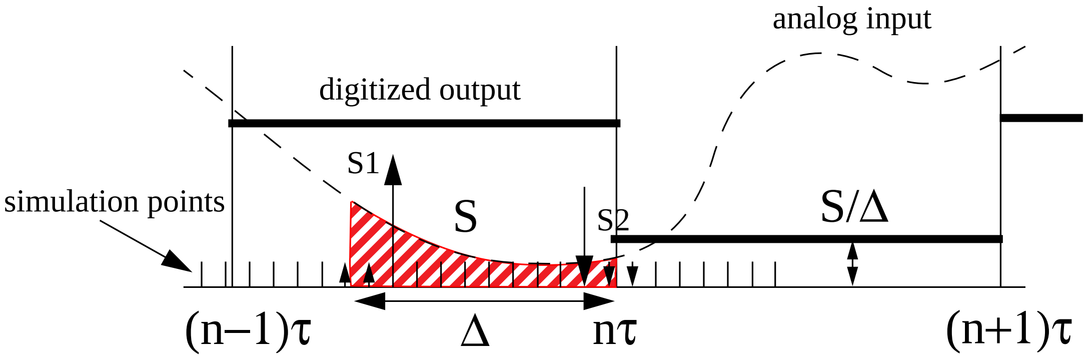


Figure 8: Digitization in Time

4.18. VAL_digitizer

First, the input value is folded in the range between min_val and max_val. Then the input value is digitized using the following formula.

$$out = \mathrm{min\_val} + floor\left(\frac{input - \mathrm{min\_val}}{D}\right) \cdot D$$

where D = (max_val - min_val) / ($2^{\mathrm{num\_bits}}$-1), and floor(x) gives the largest integer which does not exceed x.

5 FREQUENTLY ASKED QUESTIONS

5.1. 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.

5.2. 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
```

5.3. 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.

5.4. 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 recordring 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

5.5. 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.

5.6. How can I implement a phase 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:

$$\phi(t) = \int_0^t \omega(t)dt = \omega_0 \cdot t + \int_0^t \delta\omega(t) \qquad (7)$$

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.

6 DESCRIPTION FILE SYNTAX

6.1. Outline

The syntax of the description file to be supplied for the simulation program is as follows. Bold faced strings are keywords and should be typed as it is. Italic strings are primitive names and setting names thereof. The name of the instances of primitives can contain alphabets, numbers and underscore line. “box” is a kind of primitive, but it behaves differently from the rest of the primitives. Because of that, it is displayed as a keyword for the sake of clarity. When you create a box, it can be saved with a name following the rule for the naming of primitives. The box file can be included in other box files. In that case, the exisiting (included) box file behaves the same way other primitives. In the following, name “module” is used to represent “primitive” and “box” together.

6.2. Syntax

Blank lines can be inserted.

% all the rest after % is treated as comments.

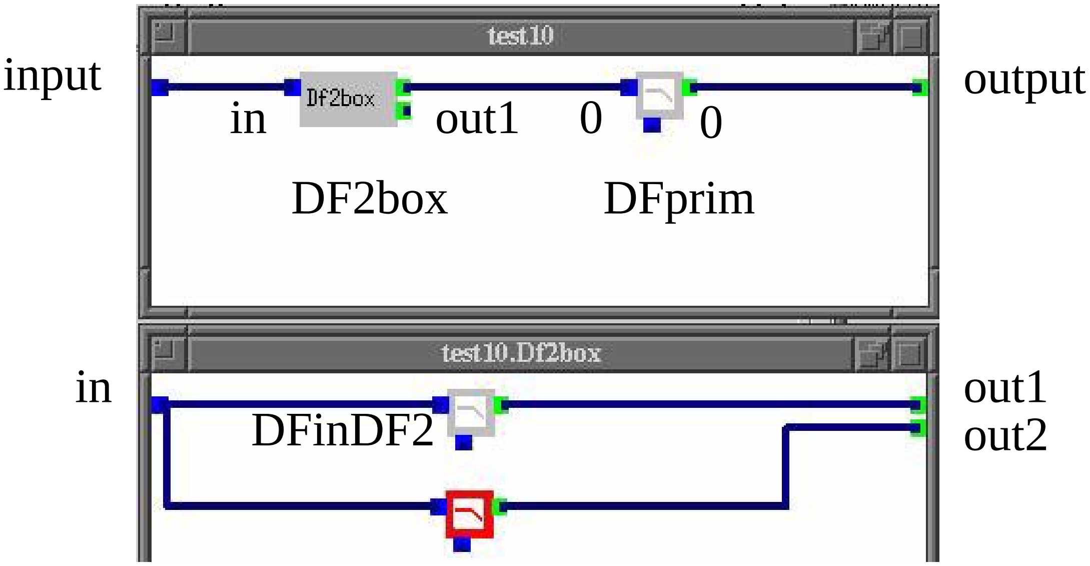
Add_Submodules
{
    primitive1 userDefinedPrimitive1
    ...
    box userDefinedBox1 { **#include** *box1* }
    ...
}

Settings userDefinedPrimitiveN
{
    setting1 = valueOfSetting1
    ...
    #include filename1

```
    ...
}
...
Settings userDefinedBoxN
{
    Settings userDefinedPrimitiveInThisBox
    {
        setting1 = valueOfSetting1
        ...
    }
    Settings userDefinedBoxInThisBox
    {
        ...
    }
}
...

Add_Connections
{
    this inPort1 -> usedDefineModule1 inPort1
    ...
    usedDefineModuleN outPort1 -> usedDefineModuleM inPort1
    ...
    usedDefineModuleL outPort1 -> this outPort1
    ...
}
```

6.3. Example



```
Add_Submodules
{
  box DF2box { #include DF2.box }
  digital_filter DFprim
}
Settings DFprim
{
  pole = -1
}
Settings DF2box
{
  Settings DFinDF2
  {
    gain = 1.0
  }
}
Add_Connections
{
  this input -> DF2box in
  DF2box out1 -> DFprim 0
  DFprim 0 -> this output
}
```

6.4. Explanation of the syntax

6.4.1. Add_Submodule

Add_Submodule section surrounded by { and } defines modules, primitives and boxs, included in this description or box file, and assign names to each of the included modules. E.g., in the exmple above, one box, whose file name is DF2.box, is included and is named as DF2box. Also included in a digital_filter primitive and is named as DFprim.

6.4.2. Settings

Settings section defines variuos settings of primitives and boxes included. In the example above, DFprim's pole is set to be -1. The setting is primitived in a box included can be defined in a similar way. In the example, gain of the digital_filter DFinDF2 in DF2box is set to be 1. You can create a separate text file and include it in the definition of the setting.

6.4.3. Add_Connections

Add_Connections section defines the data connection. A data connection is defined by a pair of ports, an output port of a module connected to an input port of another module. Two exceptions are "**this** input" and "**this** output". The current box is called **this** so that renaming does not affect the definition of the connection. They are the input and output ports of the current box, and "**this** input" is connected to an iput port of an included module and an output port of an included module is connected to "**this** output".

When the time domain simulation goes, the input data are prepared first, then it is passed to the input port of other modules, and when a module has all input data set, that module is executed to generate the output.

In the example, the input to this box is passed to the input of the box DF2box, and one of the output of DF2box, out1, is passed to a primitive DFprim, whose input port name is "0", and the output of this primitive, output port name "0", is passed to the output of this box, named "output".

6.5. alfi output

alfi output files contain extra information for its use. Those information are stored in a line which starts with "%*". Because the simulation program neglects text after %, all information for alfi are just for alfi use. These informations are the sizes of the window, the locations of links on a line linking two ports, etc. If you create a description file, or box file, and later open it using alfi, primitives and boxes will be located at the top left orner of the window, and all links will be arranged using a default (the way the smart link option would generate) rule.

APPENDIX 1 REFERENCE

[1] LIGO-T970194 "Organization of End to End Model"

[2] LIGO-T970196 "Physics of End to End model"

[3] LIGO-T990081 “Time Domain Modal Model in e2e simulation package”
[4] LIGO-T990106 “Mechanical Simulation Engine : Physics”