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

Document Type **LIGO--T000047-03 - E** Nov. 2002
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.10, 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. modeler and modeler_freq

modeler is an application to simulate the specified system in the time domain.

modeler_freq is an applciation which calculate the transfer function of the specified system in the following way.

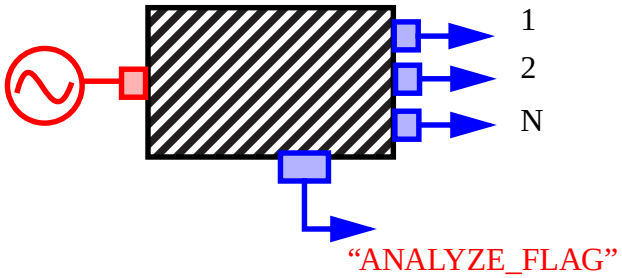


Figure 1: modeler_freq

The system to be analyzed has one input source and multiple output ports. A sinusoidal signal is suplied to the input port, $A_{in} \sin(\omega t)$. The program runs until the outputs become stable, i.e., the amplitudes of the frequency become constant, $A_i \sin(\omega t + \phi_i)$, for each output. Then the amplitude of the transfer function is calculated as A_i / A_{in} and the phase shift to be ϕ_i.

If the system is complex, it may take a long time or an unpredictable amount of time until the outputs become stable. To analyse these systems, create an output port whose name is “ANALYZE_FLAG”. When an outport with this name exists, modeler_freq analyzes the system only when this output is not zero.

E.g., if one wants to calculate transfer functions of the LIGO system, the analysis needs to be done after the field in the cavity is fully built up and is stationary. To calculate these transfer functions, create an output “ANALYZE_FLAG” which is 0 during the lock acquisition process, and set it to non zero when the powers of the two arms are larger than some threshold value (e.g., 95% of the full power). Better yet is to wait a little bit to let the field buildup fully before the flag is turned on.

2.3. 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
bundle	collection of data with names. DNToBundle is used to merge data with name to a bundle, and DNFromBundle is used to extract data from a bundle by identifying by name	multiple data passed together between boxes	adlib_bundle
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	$\sqrt{watts}$

Table 2: Units

Quantity	*Unit*
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_reader (Sec. 4.13.)	read data from a file and generates interpolated or extrapolated data series.	none	“output” vector_real	“fileName” string, numData integer (0), skipLines(0) integer, boolean useFilter (true)
data_out	used to get data out of the simulation (a "probe")	"0" variable type	none	none
data_viewer (Sec. 4.14.)	Interavively view data	"0" variable type	none	none
psd_out (Sec. 4.28.)	accumulate the input, calculate psd of the input and write to a file	“0” real “activate(1)” real	disk file whose name is the full path of this primitive	f_from (0.1), f_to (10) real logSpacing (true) boolean N_freqs (100), Lowpass_Order (6), Highpass_Order (6), SlopePower (not defined), N_TffT (1), N_delT (4), maxCounter (100) integer
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. 4.17.)	a digital filter out = digiatl filter (in)	"0" real “resetOn”	"0" real	"zero" "pole" (in rad/sec.) "gain" real "zeropair" "polepair" complex “forceQuad” boolean (false) “sampelTime”(0) real
ADC (Sec. 4.20.)	Discritize the input with the specified sampleTIme.	“0” real	“0” real	“gain”(1), “sampleTime”(0), “integrationTime”(0) real, “numBits” integer(0), “signedInt” boolean (true)
DAC (Sec. 4.21.)	digitize the input value using finite number of bits. A noise model based on bit flipping is included.	“0” real	“0” real	“gain” real(1), “numBits” integer(0), “signedInt” boolean (true), “flipTime” real(0)
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
xor	exclusive 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
flipflop	flipflop if (reset) state = false; else { if(set) state=true;}	“set”, “reset” real	“0” real	same as above initial state is false
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
hardSwitch (Sec. 4.26.)	Connect one of the inputs to the output based on the switch (ON if non-zero)	“ONinput”, “OFFinput” unknown	“out” unknonwn	switch(1) real
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

Name	*Function*	*in*	*out*	*setting*
clock	generates the time	none	"0" real	none
Unit Conversion				
lam2k	converts wavelength to wavenumber out = 2 π / in	"0" real	"0" real	none
f2k	converts frequency to wavenumber out = 2 π in / c	"0" real	"0" real	none
Type Conversion				
field2complex (Sec. 4.11.)	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”,. “FY”, “FZ”, “torqueX”, “torqueY”, “torqueZ” 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
ClampToBundle (Sec. 4.27.)	Convert clamp data into a bundle	“in” clamp	“out” bundle	“nameNN” where NN=00~12 (string) default values are the same as the output names of clamp2xyz.
Field Operation				

Name	*Function*	*in*	*out*	*setting*
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. 4.15.)	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	
fld_modulator (Sec. 4.19.)	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. 4.18.)	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 ,
beam_wiggler (Sec. 4.6.)	deviation of the beam at small angles	"0" field, “thetaX”, “thetaY” real,	“0” field	none
beam-shifter (Sec. 4.7.)	small transversal shift of the beam	“0” field, “dx”, “dy” real	“0” field	none
pd_demod (Sec. 4.16.)	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) “KeepGuoyOffset” bool (no), “dphiGuoy” 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

Name	*Function*	*in*	*out*	*setting*
telescope (Sec. 4.12.)	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:				
cav_sum [b] (Sec. 4.8.)	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) “KeepGuoyOffset” bool (no), “dphiGuoy” 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), “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 tri-angular 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); “KeepGuoyOffset” bool (no), “dphiGuoyAB”, “dphiGuoyBC”, “dphiGuoyCA” 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),

Name	*Function*	*in*	*out*	*setting*
rec_sum [c] (Sec. 4.9.)	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) “KeepGuoyOffset” bool (no), “dphiGuoyA”, “dphiGuoyB”, “dphiGuoyC” 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).
mechanics				
susp3Dmass (Sec. 4.22.)	Simple suspended 3D mass.	“suspPt”, “force” clamp	“massPos” clamp	Data type of all settings are real. “Thickness” (0.10), “d_yaw” (0.0333), “d_attach” (0.25506), “d_pendulum” (0.450), “d_CM” (0.0014), “d_pitch” (0.0082), “Mass” (10.30), “QvalInvZ” (1e-4), “QvalInvPITCH” (1e-4), “QvalInvYAW” (1e-4), “InitPosZ” (0), “InitPosPITCH” (0), “InitPosYAW” (0), “InitVelZ” (0), “InitVelPITCH” (0), “InitVelYAW” (0)
Vector operations (Sec. 4.23.)				
VecLen	the size of vector	“Vin” vector_real	"length" real	none
VecScaMerge	Append (up to 16) scalars to the input vector to make a larger vector	“Vin” vector_real, “scalar00” ~ “scalar15” real	“Vout” vector_real	“scalarDataSize” integer (-1)
VecVecMerge	Merge 2 vectors into one	"V0in", “V1in” vector_real	"Vout" vector_real	none
VecSegment	Extract part of the input vector. Vout[0...length-1] = Vin[base ... base+length-1]	"Vin" vector_real, “base”, “length” integer	"Vout" vector_real	none

Name	*Function*	*in*	*out*	*setting*
VecElem	Return one element of a vector, val = Vin[index]	“Vin” vector_real, “index” integer	“val” real	none
VecManyElems	Extracts up to 16 elements of the input vector to scalar outputs, s00=Vin[0]...s15=Vin[15]	“Vin” vector_real	“scalar00” ~ “scalar15” real	none
VecSubs	Substitute a value into one element of the input vector Vout=Vin, Vout[index]=val	“Vin” vector_real, “index” integer, “val” real	“Vout” vector_real	none
VecAdd	a*V0in + b*V1in	“V0in”, “V1in” vector_real, “a”, “b”, real	“Vout”	none
ClampAdd	a*clamp0 + b*clamp1	“clamp0”, “clamp1” clamp, “a”, “b” real	“clampOut”	none
MatInv	Mout = 1 / Min, if fails, status is set to 0. M[i,j] is (i*column size+j)th in vector M.	“Min” vector_real	“Mout” vector_real, “status” real	none
MatVecProd	Vout = Min * Vin. size of Min should be size of Vout x size of Vin. Min[i,j] is (i*Vin size + j)th in vector Min.	“Vin”, “Min” vector_real	“Vout” vector_real	none
AxisRotation (Sec. 4.24.)	Rotate the coordiante axis and calculate the vector in the new axis.	“inClamp” clamp	“outClamp” clamp	Three Eularian angles, “phi” (0), “theta” (0) and “psi” (0)real
Generic Functions (Sec. 4.23.) **If other combination of inputs and outputs are necessary, please let the developer know (e.g., 2 vectors and 2 scalars).**				
FUNC_1x1 FUNC_2x2 FUNC_4x4 FUNC_8x8 FUNC_16x16	n real input to n real output	“in0”, “in1”, ... real	“out0”, “out1”,... real	“Equations” string
FUNC_VxV FUNC_2VxV	n vector_real input to n vector_real output	“inVec0” ... vector_real	“outVec0” ... vector_real	“Equations” string “outputVectorSize_0” integer
bundle (Sec. 4.27.)				
mergeBundles	merge two bundles to one	“in0”, “in1” bundle	“out” bundle	
D4ToBundle D8ToBundle	merge up to 4 (8)data to a bundle	“inBundle” bundle “in00”, “in01”, “in02”, “in03” unknown	“out” bundle	name00, name01, name02, name03 ... (these should be set to name the input data, otherwise triggered as error)

Name	*Function*	*in*	*out*	*setting*
D4FromBundle D8FromBundle	extract up to 4 (8) data from a bundle	“in” bundle	“out00”, “out01”, “out02”, “out03” unknown	name00, name01, name02, name03 ... (these should be set to specify the data to extract, otherwise triggered as error)
VecToBundle	Add vector to another bundle	“inBundle” bundle, “inVector” vector	“out” bundle	“offset” (0) integer “nameNN” NN=00~15 (empty string) string (see Sec. 4.27. for edetails)
BundleToVec	Add or replace the input vector component by a bundle component	“inVector” vector, “inBundle” bundle	“out” vector	“offset” (0) integer “nameNN” NN=00~15 (empty string) string (see Sec. 4.27. for edetails)

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), **theta_z** (roll) and its output can be connected to “mech_data” port(s) of optics modules. All these quantities are defined in a right-handed coordinate system.

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

“**y**” & “**x**” : displacements in transverse directions, y in vertical and x in horizontal direction.

“pitch” or **“yaw”** : “pitch” is rotation around the horizontal axis, **x**, and “yaw” is rotation about the vertical axis, **y**. Consider the normal to the front (coated) surface of a perfectly aligned mirror. This is

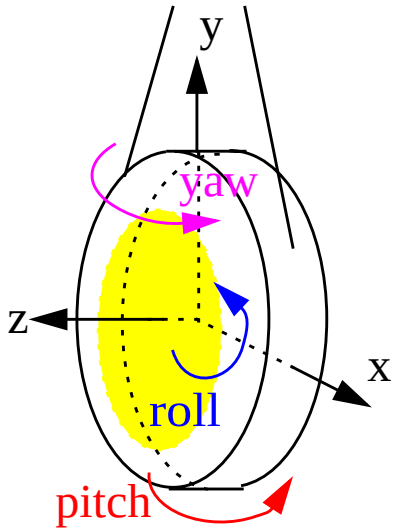


Figure 2: Definion of axis and angle

the z-axis. Now you know the positive x-axis and y-axis in a right-handed frame. The positive values of “pitch”, “yaw” and “roll” are rotation of mirror in clockwise (right-handed convention) directions around positive x, y and z axes respectively.

3.2. Definition of length between optics

3.2.1. General arguments

The lengths between mirrors are very important for the simulation of optics systems.

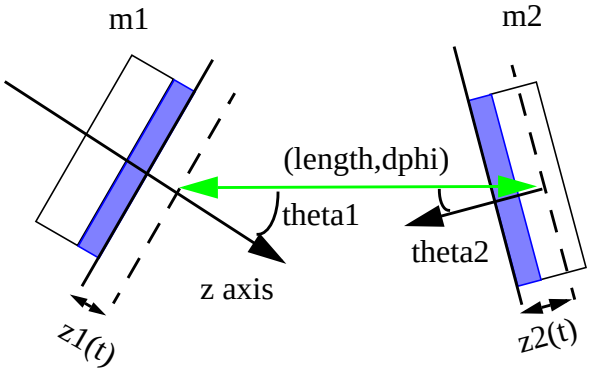


Figure 3: Definition of length

As is shown in Figure 3, the distance between the two mirrors, m1 and m2, are functions of 6 quantities, length, dphi, $z_1(t)$, theta1, $z_2(t)$ and theta2. The meanings of each parameter and an explicit formula are given in the following subsections. For each mirror, there is a reference plane, shown by dashed lines in Figure 3, which is time independent. The length between mirrors is calculated using a time independent length between reference planes of mirrors, L_{12} (calculated using “length” and “dphi” defined in primitive “propagator” and in various summation cavity primitives, like “rec_sum”) and time dependent mirror positions, $z_i(t)$, which are inputs to various optics primitives.

E.g., a Fabry-Peroit cavity is constructed by two mirrors with two progapators connecting these two mirrors, as is shown in Figure 4.

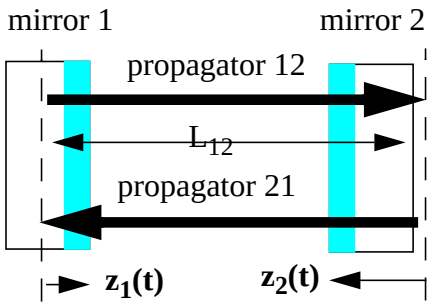


Figure 4: Fabry-Peroit cavity

The static distance, L_{12}, is defined in the two propagators, and the mirror displacement, $z_i(t)$, with respect to the reference plane, is given as a dynamic input to each mirror as the mirror motion. The distance between two mirrors at time t is calculated by

$$L(t) = L_{12} - z_1(t) - z_2(t) \quad (1)$$

The mirror displacements are subtracted because of the z axis convention, i.e., z axis of each mirror is pointing outward from the coated surface. When mirrors are displaced with respect to their respective reference plane as is shown in Figure 4, z_1 and z_2 are both positive in this convention, and the length between the two mirrors are shorter than L_{12} by z_1+z_2.

The choice of the static distance, L_{12} and the reference planes of the mirrors, is not unique. Only the summation of the static distance and mirror displacements, i.e., Eq. (1), is physically meaningful. Proper choice of a static distance makes the simulation setup easy. If L_{12} is set to be the cavity resonance length, the mirror displacements can be set to 0 to setup a resonant cavity.

3.2.2. Time independent length between reference frames - scalar field

When a scalar field propagates through a distance L, the phase changes by the following amount:

$$\phi_{scalar} = -2\pi\frac{L}{\lambda} = -2\pi N + \delta\varphi \tag{2}$$

$$\delta\varphi \equiv -2\pi\left(\frac{L}{\lambda} - N\right) \tag{3}$$

where λ is the wavelength of the field and N is an integer closest to L/λ. A field is resonant in a FP cavity when $\delta\phi = 0$ or π, and is anti resonant when $\delta\phi = \pi/2$.

In e2e, the wavelength of a field, λ, is defined using a reference wavelength, λ_0, and the offset dk as follows:

$$\frac{2\pi}{\lambda} = \frac{2\pi}{\lambda_0} + dk \tag{4}$$

λ_0 is defined in **field_gen** primitive which is the source of a field. Usually, dk = 0 for the carrier, but a nonzero value can be assigned by using **freq_shifter**. For a sideband with the modulation wavelength $\lambda_{MOD}=c/f_{MOD}$, dk = $2\pi/\lambda_{MOD}$.

In e2e, the inter-optics length is specified by a pair of values, *length* and *dphi*. *length* is used to give the macroscopic length, and *dphi* is used to specify $\delta\varphi$. In other words, *dphi* defines the deviation of the cavity length from the resonant status, $\phi_{resonance} = 2\,N\,\pi$.

If the numerical value of *length* is used literally to calculate the phase change of the field by $2\pi \cdot length/\lambda$, the numerical value of *length* needs to be specified with more than 13 digits to specify a 4km cavity to be resonant, after identifying the exact numerical number of the wave length used in the simulation. With the e2e convention, the cavity is resonant when *dphi* = 0 or is anti resonant when *dphi* = $\pi/2$, for any value of *length*.

All physics quantities can be calculated with enough accuracy using the macroscopic quantity *length* and the microscopic adjustment *dphi*. E.g., the phase change of the sideband relative to the carrier field is calculated by *length*/"wave length of RF modulation" (~10m) and the Guoy phase by *length* / "Raleigh range" (~1km).

3.2.3. Time independent length between reference frames - multi mode field

The (m,n) component of a Gaussian beam acquires the following phase when propagating through a distance of L (Ref.[3]),

$$\phi = -2\pi\frac{L}{\lambda} + (m+n+1)\eta_{00} = \left(-2\pi\frac{L}{\lambda} + \eta_{00}\right) + (m+n)\eta_{00} \tag{5}$$

where η_{00} is the Guoy phase acquired by the TEM00 mode through this propagation. For a field which is the TEM00 eigenstate of a FP cavity of length L with mirror curvatures of R_1 and R_2, the Guoy phase change propagating through the cavity is given by the following equation.

$$\eta_{00}^{FP} = \mathrm{acos}\left(\sqrt{\left(1-\frac{L}{R_1}\right)\left(1-\frac{L}{R_2}\right)}\right) \tag{6}$$

The phase change of the (0,0) mode can be rewritten in the following way.

$$\phi_{00} = -2\pi\frac{L}{\lambda} + \eta_{00} = -2\pi N + \delta\varphi + \delta\eta \tag{7}$$

$$\delta\varphi \equiv -2\pi\left(\frac{L}{\lambda} - N\right) + \overline{\eta_{00}} \tag{8}$$

$$\delta\eta \equiv \eta_{00} - \overline{\eta_{00}} \tag{9}$$

which is a generalization of Eq.(2) and (3). Here, η_{00} is the Guoy phase change dependent on the field quantity and $\overline{\eta_{00}}$ is a constant setting named *dphiGuoy*, in the propagator and other summation cavity modules. When a boolean flag *KeepGuoyOffset* is false, which is the default, *dphiGuoy* setting is neglected and $\overline{\eta_{00}}$ is set to be η_{00}.

Just in the same way discussed for the scalar field, the cavity distance are specified by *length* and *dphi* (Eq.(8)), and the phase of the (m,n) mode with dk offset (see Eq.(4)) is calculate by

$$\phi(m,n,dk) = -2\pi N + (m+n)\eta_{00} + \delta\eta + dk\cdot length + dphi \tag{10}$$

When *KeepGuoyOffset* is false, $\delta\eta$ =0, and, it is easy to setup a cavity where TEM00 carrier (m=n=dk=0) component of the incoming field is a well defined state. But, with this convention, the length of the cavity implied by *dphi* = 0 changes as the mode base[1] of the field changes. The size of the change of the length is very small, but this change can be important when discussing locking.

A good example is the thermal lensing effect of the input test mass.[2] Imagine a case that a TEM00 field with a given mode base is going into a FP cavity. The field in the cavity has a different mode base than the input field due to the input test mass. Another way to say is that the curvature of the field changes when the field goes through a lens. This change of base is automatically calculated

1. A Hermite-Gauusian field is characterizewd by two independent parameters (see [3]). In e2e, the waist sise and position are chosen to define the base of the mode, as can be found when specifying the field in **field_gen** primitive. In the following discussions, “change of the mode base” means the change of these parameters.
2. Only the lensing effect due to the radius dependent refractive index change of the substrate is discussed here.

by e2e using the refractive index of the mirror. As the input test mass is heated up, this lens effect changes, and accordingly the mode base of the field into the cavity changes. [3]

This effect can change the mode base in the cavity large enough that the value of η_{00} can change to cause measurable effect. If the cavity length is defined with *KeepGuoyOffset* false and *dphi* = 0 (or any constant), this effect is not simulated, because the change of the Guoy phase of the carrier is automatically compensated by the change of length.

In order to simulation this kind of effect, *KeepGuoyOffset* needs to be set to true, and some value is to be assigned to *dphiGuoy*, and the same setting should be used to simulate different states of interest, like cold and hot states of the input test mass. For a FP cavity, using the Guoy phase change determined by a cavity geometry, Eq.(6), for a cold state, will be a good choice.

Since the choice of *dphiGuoy* is arbitrary, the use of this setting leaves the chore of finding the resonance point of a cavity to the user. This is usually acomplished with a control loop and, and the ability to observe of the action of this loops is likely the motivation for setting *KeepGuoyOffset* to true.

3.2.4. Time dependent mirror displacements

The mirror position, which can be time dependent, is defined to be the relative distance between the mirror surface and the mirror reference plane, using the perpendicular direction pointing outward from the coated side (shown by a gray box in Figure 3) of the substrate as the axis. So in Figure 3, z_1 is negative while z_2 is positive. The effect of the mirror displacement, z_1 and z_2, are taken into account by the change of the phase. As is shown in Figure 5, the net change of the path length is $2z \cdot \cos\theta$, and the phase change due to this difference is added to the reflected field.

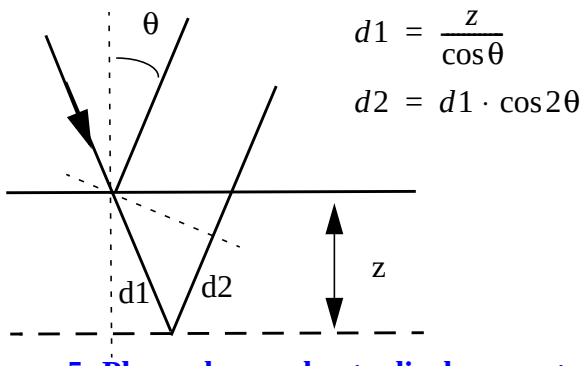


Figure 5: Phase change due to displacement

4 PRIMITIVE MODULES

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 utilized 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.

3. The thermal lensing effect can be simulated in a crude way by changing the refractive index of the input test mass.

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 categories:

4.1.1. simulation information:

"max_mode_order": represents the maximum order (m+n of TEM) up to 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 transverse dimensions). Setting "max_mode_order" to zero or other positive integer (up to 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 up to 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 up to 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”** : In the plane wave case (when you select “max_mode_order” = -1 in “field_gen” module of your .box file) , the total length of any propagation path is calculated as follows:

$$L_0 = \left(N\left[\frac{length}{\lambda}\right] + \frac{dphi}{2\pi} \right) \cdot \lambda \qquad (11)$$

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

"**KeepGuoyOffset**" and "**dphiGuoy**" : See Section 3.2.3.

"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:

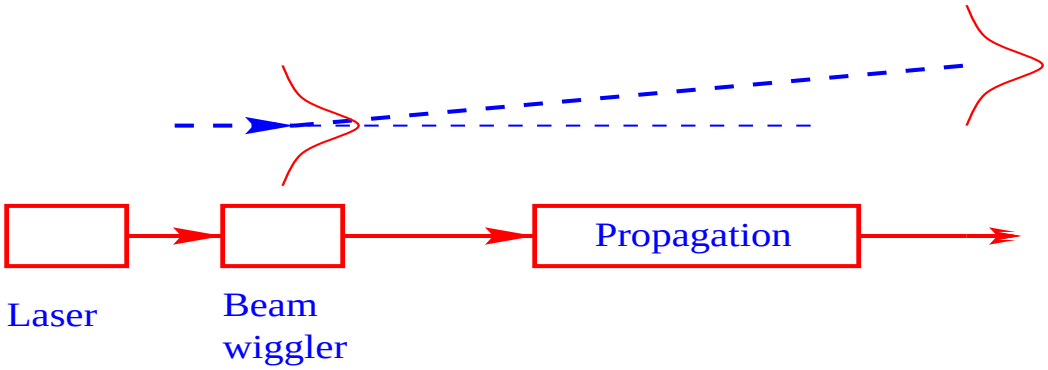
Module removed. Use telescope instead.

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" : T**o 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. beam-wiggler:

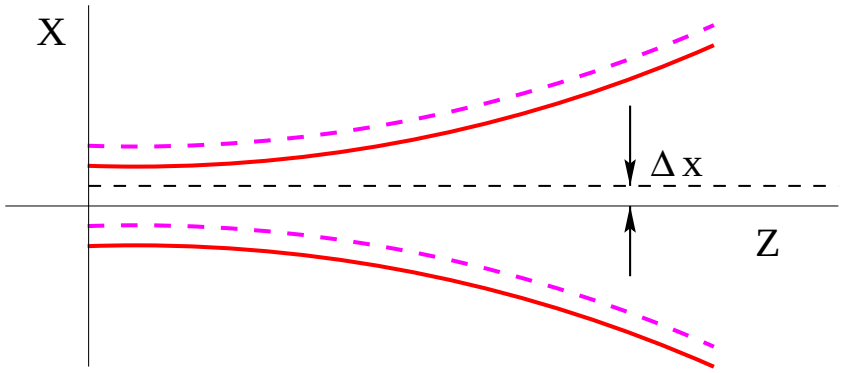
This can rotate the beam around horizontal “x” axis (“thetaX”) or vertical “y” axis (“thetaY”) by small angles (as compared to the divergence angle of the beam). For example, as shown in figure,



if the ‘beam-wiggler’ module is put on a beam path and appropriate values of “thetaX” and/or “thetaY” are set to it, the beam direction will rotate by the specified angles. If the beam propagates some finite distance after that, we can see that its maximum power position in transverse direction moves some finite distance from the center. One should note that in this particular case, while propagating, the effect of the addition of gouy phase is the only important one for the angular deviation of the beam to happen. The time-delay of propagation is not important. So, if one is using this set-up with some other cavity, one may like to set the time-step appropriate for the cavity without bothering about the time-delay for this propagation. One may do this by using either the “prop” module with “have_delay” off or using “telescope”.

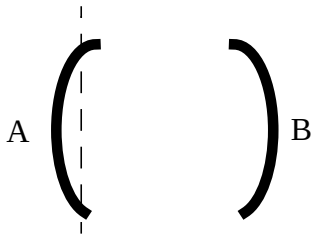
4.7. beam-shifter:

This can shift the beam in transverse “x” (horizontal) or “y” (vertical) directions by small amount (as compared to beam waist size).



4.8. 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.



virtual pickoff (C,Apick)

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).

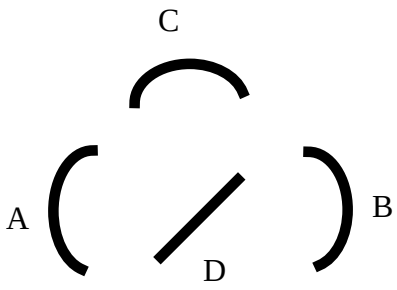
Lensing effects of the component mirrors have been included in calculations. So, do not forget to set **“refractive_index”**, **“radius_front”** and **“radius_back”** of mirrors A and B.

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

“dphi” is longitudinal phase offset. One may use another phase offset **“dphiGuoy”** after setting the boolean flag **“KeepGuoyOffset”** to true or yes. The offset “dphiGuoy” is useful for comparing, say, simulation runs with various levels of mode matching in a cavity to find out absolute values of changes in mirror positions in these runs by setting “dphiGuoy” to a reference value (See Section 3.2. and Section 4.3. for detailed explanation).

4.9. 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.

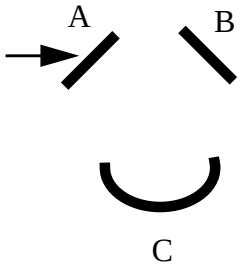
Give mechanical data of the mirrors through "**mech_dataA**", "**mech_dataB**", ... ports (see section 3 "Convention"). Remember: the longitudinal position, z, of the beam-splitter refers to shift along the normal to its coated surface, just like in any other mirror (and the $\sqrt{2}$ factor is taken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

"dphi"s are longitudinal phase offsets. One may also use a set of other phase offsets **"dphiGuoyA","dphiGuoyA", "dphiGuoyA"** after setting the boolean flag **"KeepGuoyOffset"** to true or yes. The offsets "dphiGuoy"s are useful for comparing, say, simulation runs with various levels of mode matching in a recycling cavity to find out absolute values of changes in mirror positions in these runs by setting "dphiGuoy"s to a reference set of values (See Section 3.2. and Section 4.3.for detailed explanation).

4.10. 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.

One may also use a set of other phase offsets **"dphiGuoyAB", "dphiGuoyBC", "dphiGuoyCA"** after setting the boolean flag **"KeepGuoyOffset"** to true or yes. The offsets "dphiGuoy"s are useful for comparing, say, simulation runs with various levels of mode matching in a cavity to find out absolute values of changes in mirror positions in these runs by setting "dphiGuoy"s to a reference set of values (See Section 3.2. and Section 4.3. for detailed explanation).

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

4.11. field2complex:

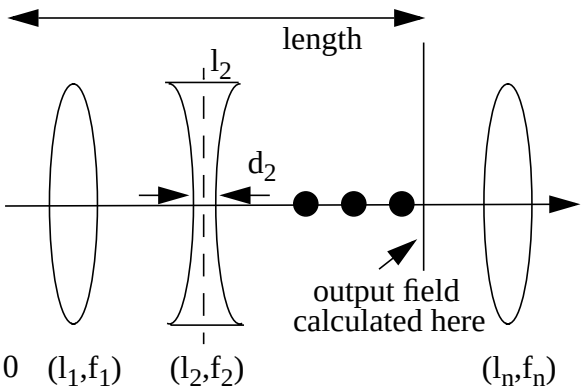
This module allows one to get the complex amplitude of a field (which, by E2E construction, is a class containing various field information and associated functions) in frequency specified by

"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"**.

4.12. 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.

4.13. data_reader

Read data from a file "fileName" and interpolate or extrapolate to generate time series of data. The input file should have the time in the first column and arbitrary number of columns of data, all separated by white spaces. A data series in each column is interpolated or extrapolated using a 2nd order polynomial. This output is filtered by a lowpass filter (Chebyshev 2, 10th order, 40dB, Nyquist frequency determined by the first two input times), if "useFilter" is true. All outputs are stored in a vector output.

If "numData" is positive, first "numData" data series are processed. First "skipLines" lines are skipped and all lines are skipped which do not start with a number (starts with a character which is not a digit nor period).

4.14. 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.

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

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

$$\begin{aligned} E_{out} &= E_{in} \cdot Exp(i\Gamma_{\varphi}\sin(\Omega t)) \cdot Exp(\Gamma_{amp}\sin(\Omega t)) \\ &= E_{in} \cdot \sum_{N=-\infty}^{\infty} \left(\sum_{i=-\infty}^{\infty} (-i)^{N-i} \cdot J_i(\Gamma_{\varphi}) \cdot I_{N-i}(\Gamma_{amp}) \right) \cdot Exp(iN\Omega t) \end{aligned} \quad (12)$$

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:

$$E_{out} \approx E_{in} \cdot \sum_{N=-n}^{n} \left(\sum_{i=-n}^{n} (-i)^{N-i} \cdot J_i(\Gamma_\varphi) \cdot I_{N-i}(\Gamma_{amp}) \right) \cdot Exp(iN\Omega t) \qquad (13)$$

4.16. 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 5 with infinite radius.

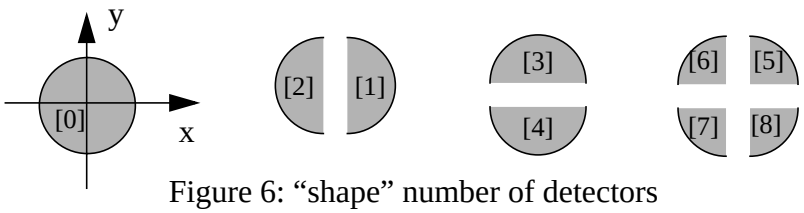


Figure 6: "shape" number of detectors

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 6, "+" and "-" signs indicate that they are added together with

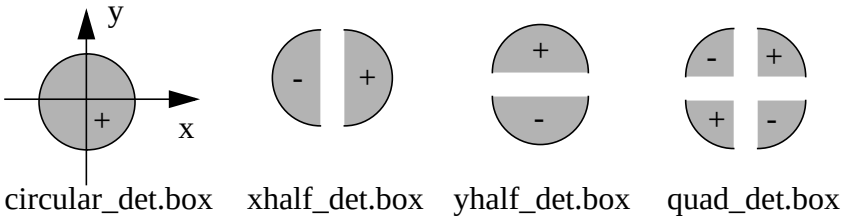


Figure 7: detector boxes

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.

Using “detmap”, you can define a detector by specifying the following quantities (see Figure 7).

- 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.

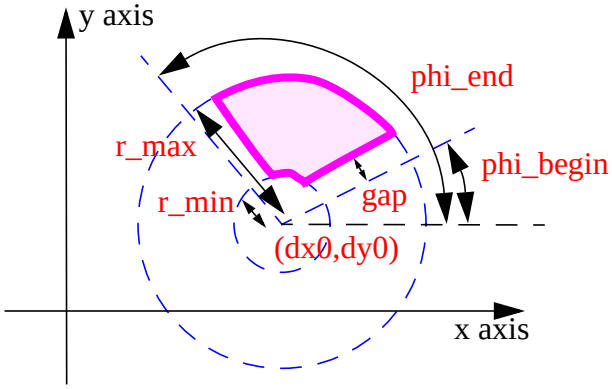


Figure 8: Specification of a detector

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.

4.17. digital_filter

The digital filter implementation in e2e is based on the same algorithm used in pziir.m by p.fritschel, i.e., (1) use bilinear transformation from s to z, (s = 2/T (z-1)/(z+1)), followed by zp2sos group-ordering implemented in matlab (DIR_FLAG = 'UP'). The filter code is based on iir_filter in ascFilter.c by R. Bork.

The specification of the digital filter is as follow:

$$DF(s) = gain \cdot \frac{f_1(\vec{z}) \cdot f_2(\overrightarrow{zp})}{f_1(\vec{p}) \cdot f_2(\overrightarrow{pp})} \qquad (14)$$

$$f_1(\vec{x}) = \prod (s - x_i)$$
$$f_2(\overrightarrow{xp}) = \prod (s - xp_i) \cdot (s - \overline{xp_i}) \quad (15)$$

The numerator and the denominator are represented by two forms of polynomials. zeros and poles in the form f1 in Eq.(5), complex zero pairs and poles pairs in the form f2. Each one is specified by a real vector (zeros and poles) or by a complex vector. The coefficients 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 "**forceQuad**" to true.

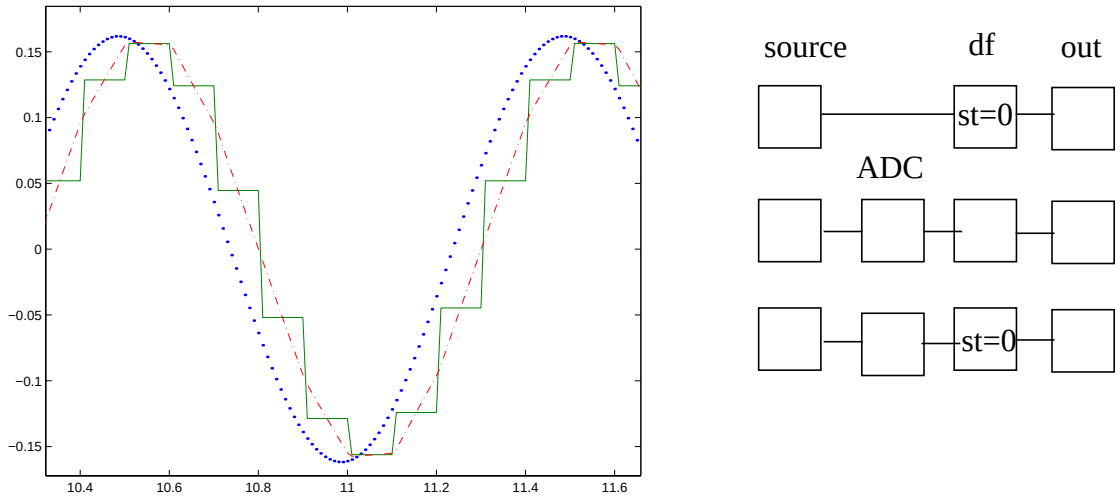


Figure 9: Digital Filter

"**sampleTime**" larger than the simulation time step (tick time), this module uses this value as the digitization time step. In Figure 8, 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.

"resetOn" switch is used to clear the internal buffer. When the resetOn values changes to a non zsero value, the internal buffer is cleared and output value is also set to 0. When resetOn is positive, the output is 0, while the evolution goes on if the resetOn is 0 or negative. The normal use is to set "resetOn" to 0. If the resetOn is set to 1, then the internal buffer is cleared and the output is 0 until the resetOn is set to 0.

4.18. 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.

4.19. 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.

4.20. ADC

For a given discritization time period "sampleTime" τ and an integration time Δ, the output between $n\tau$ to $(n+1)\tau$ is calculated as

$$out(n\tau) = \frac{1}{\Delta}\int_{n\tau-\Delta}^{n\tau} input(t)dt \qquad (16)$$

When Δ is 0, the inpuit value at time $n\tau$ is used as the output value between 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.

After this output is multiplied by "gain", the value is digitized using values between $-2^{numBits-1}$ to $2^{numBits-1}-1$ for signedInt = true or between 0 to $2^{numBits}-1$ for signedInt=false, i.e.,

$$ADC(n\tau) = floor(gain \cdot out(n\tau) + 0.5) \qquad (17)$$

and the value is bounded by the upper and lower limit of values available by an integer with numBits bits.

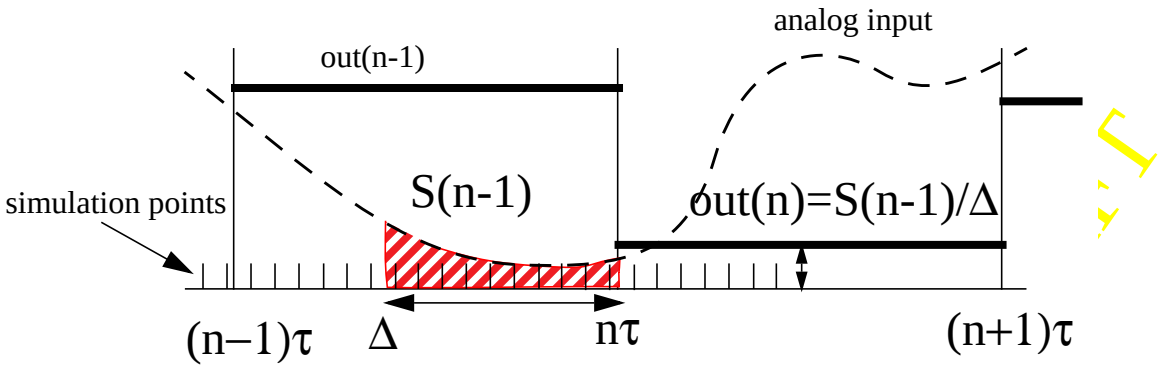


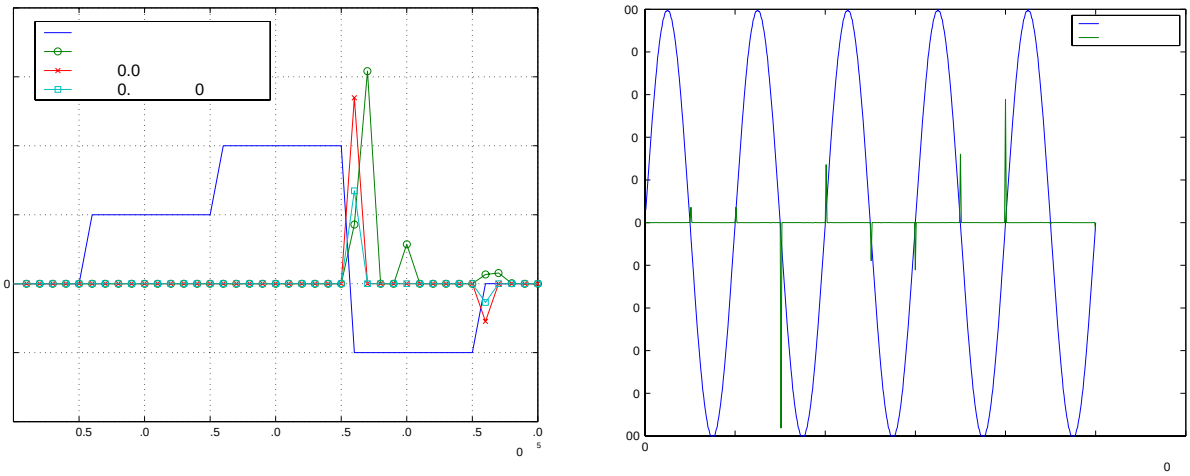
Figure 10: Digitization in Time

4.21. DAC

First the input value is digitized using values between $-2^{numBits-1}$ to $2^{numBits-1}-1$ for signedInt = true or between 0 to $2^{numBits}-1$ for signedInt=false. Then the value is multiplied by gain.

A noise model based on the bit flipping is implemened. When the digital value is changed from the previou value to the new value, some bits are flipped. The model assumes that each bit flips with an avarage time of flipTime, dN/dt = exp(-t / flipTime). The analog value is calculated as the weighted average of intermediate digital values.

For a 4 bit system, the process to changes from -1 (1111) to 0(0000) goes as follows. For each bit, a flip time is calculated, and in that order, the digital value is changed. It can be (1111)->(0111)->(0101)->(0100)->(0000). Then the analog value is calculated as (-1)xT(-1)+7*T(7)+5*T(5)+4*T(4)+0*T(0), where T(I) is the fraction of period the digital value is I. When the flipTime is negligibly small, only the last one, 0*T(0), dominates, but with finite value of “flipTime”, this can induce observable size of noise, especially when the sign bit changes.



(a) step function input **(b) sinusoidal input**

Figure 11: DAC bit flip noise

If numBits is 0 or flipTime is 0, the noise is not calculated.

4.22. susp3Dmass

This module simulates the motion of a single suspended mass based on the formulation given in Ref.[5]. The naming of settings of this module follows the one in Ref.[6] as much as possible.

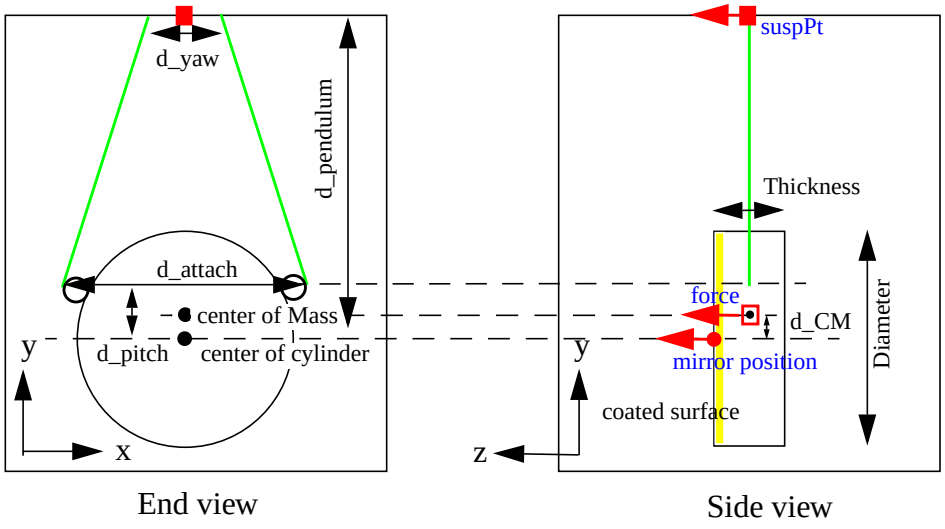


Figure 12: Single Suspended Mass

The coordinate system is defined in Fig. 2. The inputs are the suspension point ("suspPt") location and the orientation and the force and torque acting on the mirror ("force"). The output is the location and the orientation of the mass ("massPos"). The force and position are passed using clamp data type.

The origine of the coordinate systems of the suspension point and the mirror position are different. The origine of suspPt is the filled squared box in Fig. 10, while that of the mass is at the filled circle in the same figure, which is located at the center of the cylinder on the coated surface. The suspPt is located at (0, d_pendulum, -Thickness/2) in the mirror position coordinate.

This module does not include the wedge angle.

4.23. Vector operations

These modules are provided to manipulate vectors. Using these operators, one can build a vector by combining scalars or vectors or extract components of a vector.

4.24. AxisRotation

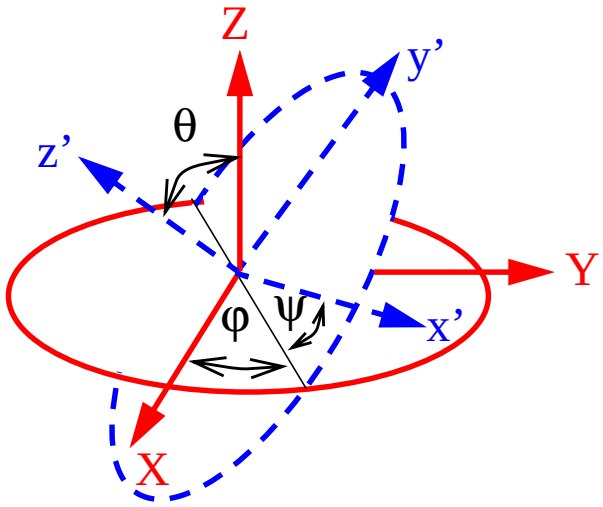


Figure 13: Euler angles

The input to this module is a clamp data (vectors) represented using one coordinate axis, X-Y-Z in Fig. 11. This module calculates the components of this clamp using another axis, x'-y'-z'. The two coordinate systems are related by three angles, phi (φ), theta (θ) and psi (ψ). First, the axis system is rotated around the z axis by φ, then by θ around the new x axis, then by ψ around the new z axis. The inverse of a transformation specified by (φ, θ, ψ) is a transformation specified by (-ψ, −θ, −φ).

The ground motion is best represented in the detector coordinate system, x axis along the x arm, y axis along the y arm and the z axis normal to the ground. The z direction of e2e coordinate system is the direction of the coated side of the mirror, and the y axis is pointing upward. Samples of AxisRotation parameters are shown in Fig. 12.

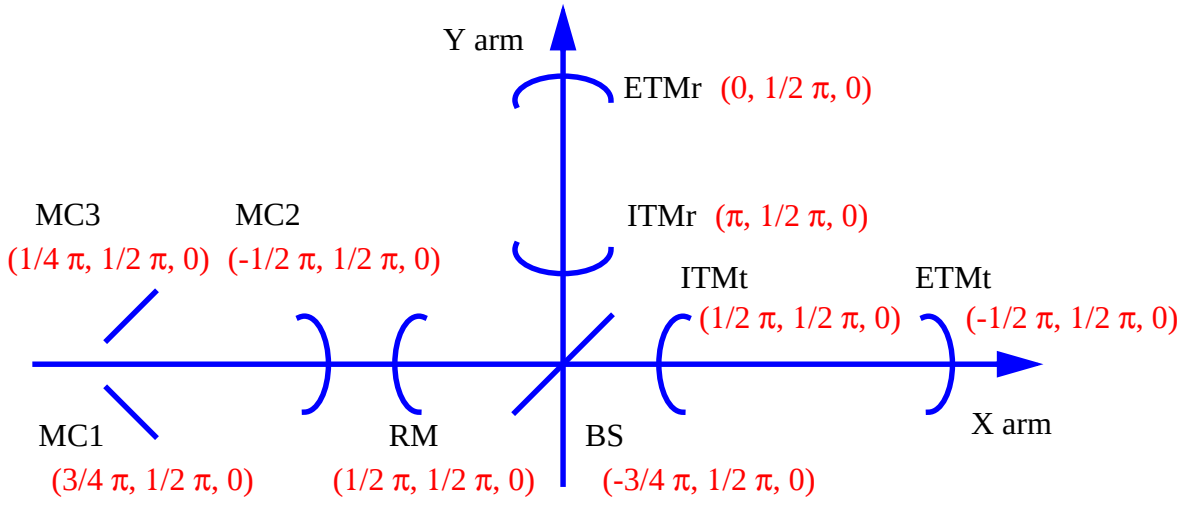


Figure 14: Detector to e2e coordinate transformation

4.25. Generic function modules, FUNC_xxx.

4.25.1. What are FUNC_xxx modules ?

These primitive modules can be used to implement mathematical calculations by using formulas with a syntax similar to the C language. E.g., by specifying the setting “Equations” of a FUNC_1x1 module to

```
out0 = sqrt(in0);
```

this module calculates the sqrt of the input value. These modules make it easier to construct mathematical calculations which are very tedious to build using primitive math modules. Some examples are given in Section 4.25.5.

4.25.2. Basic syntax

The module definition is consisted of multiple equations given in the “Equations” setting string. Each equation is of the form

```
variable name = expression using inputs, outputs (if defined), global and local
variables;
```

An identifier, name of a variable, function or macro, can be consisted of any number of alphabets, digits and underlines, except that the first character should not be a digit. Identifiers are case sensitive. Global variable, discussed in Section 5, can be used in the equation. All variables and functions are of type real.

Each equation is terminated by “;” as is the syntax of C language, except for declaration lines which start with “#”. “#define” declaration can be used to assign a string to an identifier, like

```
#define in3 inVec0[3]
```

All occurrence of in3 in the rest of the code is replaced by inVec0[3].

The names of the inputs and outputs are referred to by the names of the ports. If a port is a vector, a pair of square brackets is used to reference an element, like inVec0[1] and outVec0[2], and like. The inputs and outputs can be referenced by meaningful names by using the following declarations.

```
#inputs initial_position velocity
#output position_now
position_now = initial_position + velocity*time_now();
```

Comments can be inserted by surrounding the text by `/*` and `*/`.

4.25.3. Constants

In FUNC modules, the following constants are defined:

- TIME_STEP : simulation time step

4.25.4. Build-in functins and digital filter

The following operators are supported. The functionality and the precedence are the same as those defined in C.

- unary operators : `+, -, !`

- binary operator : `*, /, +, -, <=, >, >=, ==, !=, &&, ==`

All the standard functions defined in C and several special functions are available.

- unary functions : `sqrt, sin, cos, tan, acos, asin, atan, log, log10, exp, sinh, cosh, tanh, fabs, ceil, floor`
- binary functions : `pow, atan2, fmod, max, min`
- special functions : `hermite( n, z ), jbessel( n, z ), time_now()`
- optics functions : `fp_rayleigh_range(L,R1,R2), fp_dist2waist(L,R1,R2), fp_guoyphase(L,R1,R2), ext_rayleigh_range(z0,z,nind), ext_dist2waist(z0,z,nind)`
  : Red solid lines are for the resonant field in this FP cavity, while dashed blue lines are for the field coming in from the left mirror whose refractive index is nind. This incoming field is to match with the FP resonant field after passing the left mirror. The blue dashed lines in the FP shows the extraplation of the incoming field when there is no left mirror.
  z is the distance between the left mirror to the waist position of the cavity resonant field, while z' is that to the waist position of the out side field. For the configuration (i.e., concave seen from inside), R1 and R2 are positive, and z and z' are negative. fp_guoyphase is the total phase change of the field due to the Guoy phase propagating from one mirror to another.

NEED BETTER DESCRIPTION

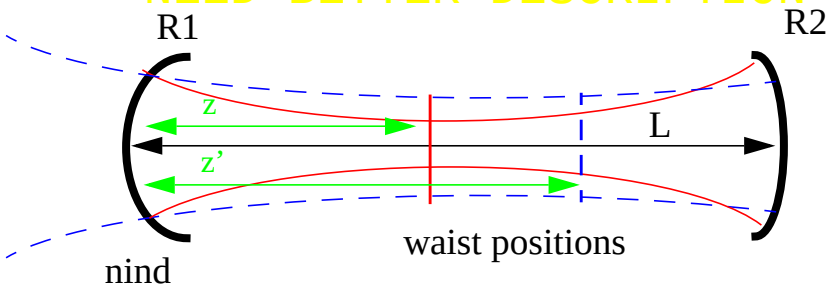


Figure 15: optics functions

- random number : `rndflat(), rndnorm(), poisson( mean ), white_noise( amplitude )`

Function "if" can be used for a conditional calculation. It takes 3 arguments :

```
val = if( condition, value_for_true, value_for_false )
```

This function returns `value_for_true` if `condition` is true or non-zero and returns `value_for_false` if `condition` is false or 0.

The digital_filter can be used in the module in the following way.

```
df(x, reset) = digital_filter( gain, {zeros}, {poles}, {zeroPairs}, {polePairs}, time_step);

out = df(in, resetVal);
```

The first line is the declaration that "df" represents a digital filter which is specified by the same parameters as the digital_filter primitive (see Section 4.17.). If the last parameter, time_step, is specified, this is used as the digitization time step, otherwise, the simulation time step is used. This parameter is to be set when this FUNC is used between ADC and DAC modules. The second dummy argument, `reset`, is optional. If no values are to be assigned for some of the zeros or poles vector, leave them empty, either like ", {}," or ", ,". There can be multiple digital filters declared in one FUNC module. When there are several digital filters used with the same specifications of gains, zeros and poles, one declaration is needed for each use.

4.25.5. Local variables and local functions

Local variables can be used without any special declaration, except that one can be used in the calculation after it's value is assigned.

```
lambda = 1.064e-6;
phi = 2*PI*input/lambda;
```

Two local variables are used here (input is the name the input and PI is a global variable name). The first declaration is parsed only once, and there is no speed penalty using these declarations. So it is a good idea to write the following kind of codes for the ease of readability.

```
gainXXX = 100; /* this is the gain by xxx */
gainYYY = 0.345; /* this is the gain by yyy */
...
totalGain = gainXXX * gainYYY * ...;
```

A local function, which is recognized only in this module, can be defined in the following way:

```
funcName( var1, var2, ..., varN ) = expression using var1, var2, ... varN and all
other local and global variables;
```

The names of function (funcName) and dummy varibales (var1, var2, ..., varN) can be any legal identifier expressions, which have not been defined for any other use. The function may have any numbder of arguments, and those arguments can be used in the definition body on the right hand side. Any example will be

```
length(x,y) = sqrt( x*x + y*y );
```

Once defined, the local function can be used in the following equation codes just as the same way as the built-in function.

4.25.6. Examples

4.25.6.1 Fiddle

```
/* the inputs and outputs are accessed using the following names */
#inputs speed offset amp freq noiseAmp reset
#outputs out linear rotation random
/* definition of digital filters */
/* there are no complex poles or zeros, and they can be omitted completely */
velocity_integrator(v,r) = digital_filter( 1, {}, {0} );
angule_integrator(v,r) = digital_filter( 2*PI, {}, {0} );
lowPass(v,r) = digital_filter( 2*PI, {}, {-2*PI} );
/* various motions */
/* first port, out, is the total motion, while 2nd to 4th port, are the individ-
ual mototions */
linear = offset + velocity_integrator( speed, reset ); /* linear motion */
rotation = amp * sin( angule_integrator( freq, reset ) ); /* rotation */
random = lowPass( white_noise( noiseAmp ), reset ); /* noise */
out = linear + rotation + random; /* total motion */
```

4.26. hardSwitch

This module has two inputs of type unknown, and one output of type unknown. If the value of switch is non-zero, then the source to “ONinput” is connected to the sinks connected from the “out” port, and the connection to “OFFinput” is disconnected (see below). If switch is zero, the input to “OFFinput” is connected to the output sinks.

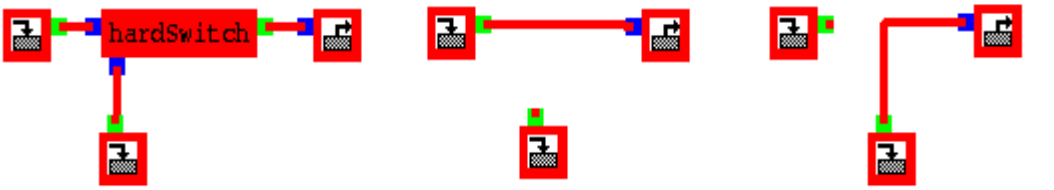


Figure 16: hardSwitch in action

Important thing to know is that the unused input connection is disconnected. In Figure 16, (a) is how the connection is setup, and (b) is how it behaves when switch is non-zero. I.e., the source to the unnecessary input port may not be executed if the output of that source is not used by other modules. If a primitive “switch” is used instead, both connections exist, both sources are executed, and one of the value is used.

There are several cases primitive “hardSwitch” is useful. One is the case that you prepare a seup in which some of the connections are connected or unconnected corresponding to different hardware operation state. Anoter is a case that the source is a time consuming module, and want to disable if not needed.

Because the data type of ports are not specified, any ports can be connected to inputs and from output of this module. But the type of the selected input and the output should match, and the type micmatch is detected as error.

hardSwitch can be used to serve as a poorman’s junktion, which will be useful unitl the junktion is supported in alfi.

4.27. bundle

When multiple related data are passed around between modules to modules, it is cleaner if those data are combined to one, and, when necessary, one can extract necessary data from that combined data stream. E.g., a box representing the core optics can have multiple fields going out. If one wants to have more going out, by adding pickoff or by adding signal recycling mirror, the box interface needs to be changed. If “bundle” data is used to interface to outside, it is only necessary to merge the new field to the out going bundle data stream.

A bundle can be thought of as an array of data with a name tag for each data. There are several kinds of primitives are provided to construct bundles and extract data from bundles.

A bundle cannot have data with same name. When merging bundles and data, a name collision is tested and, an error message is issued when detected. When extracting a data stream from a

bundle by specifying the name tag, if a data with that name does not exist in the input bundle or the input is not connected to any source, the connections from the output with the name tag are disconnected, and an warning is issued.

A bundle can contain other bundles as its components. To clarify the discussion, a word "primitive" data is used to represent data other than the bundle data stream. When data stream is extracted from a bundle, the stream can be specified by "bundle1.bundle2.dataName", or "*.bundleN.bundleM.dataName". ??? Put description here. ???

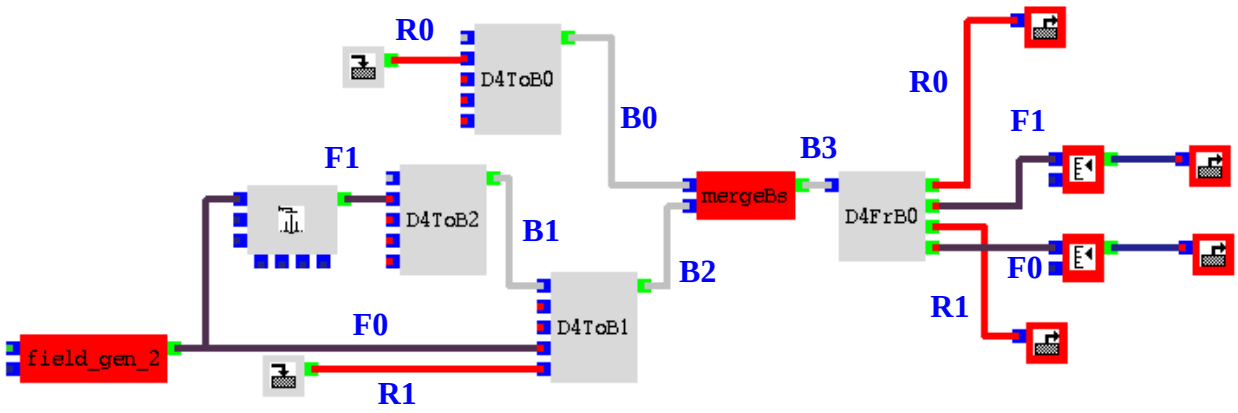


Figure 17: data flow using bundle

In this example, D4ToB0 is used to create a bundle B0 which contains one real data, whose name is assigned like "name00" = "R0". D4ToB2 is used to create a bundle which has one field data. D4ToB1 is used to merge a field and a real value to bundle B1, to create a bundle B2. mergeBs is used to merge two bundles B0 and B2 to create a new bundle B3. B3 has 2 real data and two fields whose names, say "R0", "R1", "F0" and "F1", are assigned when they first merge to a bundle, i.e., D4ToB0, D4ToB1 and D4ToB2. D4FrB0 is used to extract data from a bundle. In the example above, the settings to specify the data for each output are : "name00" = "R0", "name01" = "F1", "name02" = "R1", "name03"="F0".

4.27.1. mergeBundles

This primitive merges two bundles into one, like mergeBs in Figure 15. The name conflict is tested to avoid to create a bundle containing two same names.

4.27.2. D*N*ToBundle

There are a set of primitives named D*N*ToBundle, where N is an integer number, like D4ToBundle. This is a primitive to merge data to a bundle with 1 input for an incoming bundle and N inputs of any kind (type unknown) of data other than bundle type.

To create a new bundle, this module can be used without a source to the input bundle. The names of data are defined using settings, "name00", ..., "nameN-1". The name is tested against causing name conflict, and when detected, an error message is issued.

4.27.3. D*N*FromBundle

There are a set of primitives named D*N*FromBundle, where N is an integer number, like D4FromBundle. This is a primitive to extract data from a bundle. There are one input for an incoming bundle and N outputs of any kind of data other than bundle type. The output data for a specific port is defined by the name in the setting, “name00”, ..., “nameN-1”. The type is also checked if the sink of an output is not of type unknown, and if the type in the bundle does not match with the outgoing data type, an error is issued.

If data with the specified name does not exist in the incoming bundle, the link from the output of with that name to all sinks are removed after issuing a warning.

4.27.4. BundleToVec

Real data in the input bundle are added or substituted to the input vector, or a new vector is created from an input bundle.

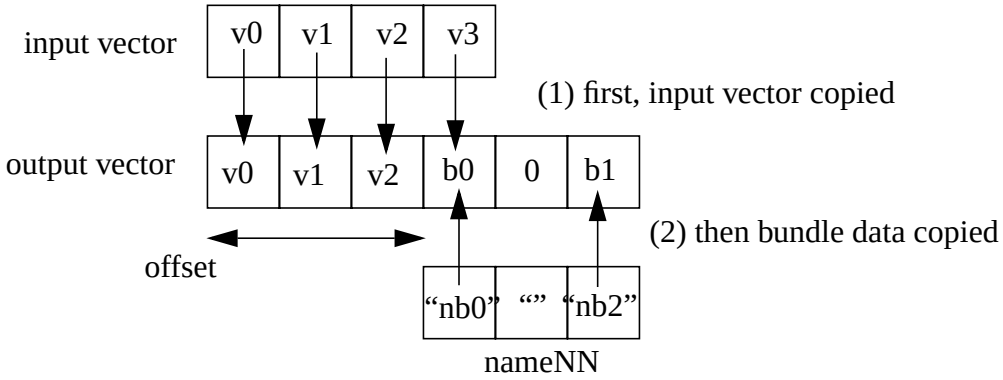


Figure 18: BundleToVec

First, the input vcector is copied to the output vector. If no input vector is specified, an empty vetor is created as the output. Then, if “nameNN” is not an emoty string, then the value of the datum in the bundle with that name is copied to the NN+offset’th element of the output vector. If necessary, the output vector is expanded by filling 0 in those elements which are not specified.

In the above example, a vector with 4 elements is created and the input vector is copied. name00 is “nb0” and name02 is “nb2” and all others are empty. The value of offset is 3. First, the value for “nb0” in the bundle is copied to the 4th element of the output data overriding the input vector, then the value for “nb2” in the bundle is placed in the 6th component of the output vector after expanding the output vector size to 6.

4.27.5. VecToBundle

Elements in a vector is merged into a bundle. If the setting of nameNN is not an empty string, then NN+offset’th element in the vector is inserted to the input bundle with the name given in nameNN. NN+offset should be less than the size of the input vector.

4.27.6. ClampToBundle

The input clamp data are placed in the output bundle. The 12 settings nameNNs are used as the names of data in the bundle. If any of the strings are empty, those elements are not copied into the output bundle. E.g., in order to build a bundle with only position information, then keep only the first 3 names, and make the rest of the 9 settings to be empty strings.

4.27.7. How it works

After all links are established, data types are checked. This is initiated by “resolve” routines of output modules. During this process, bundles are bundle related primitives modules are removed, and all links are replaced by links between normal (non bundle data) sources (inputs to D***N***ToBundle) to normal sinks (sinks from outputs of D***N***FromBundle).

4.28. psd_out

4.28.1. outline

This primitive module calculates a single sided power spectral density (psd) of the in coming real data. Based on the frequency range and resolution defined in the setting, an optimal time step ΔT (ΔT = integral multiple of the simulation time step τ) and a duration of simulation (duration = NFFT x ΔT, NFFT = power of 2) for one FFT is calculated. (See below for details).

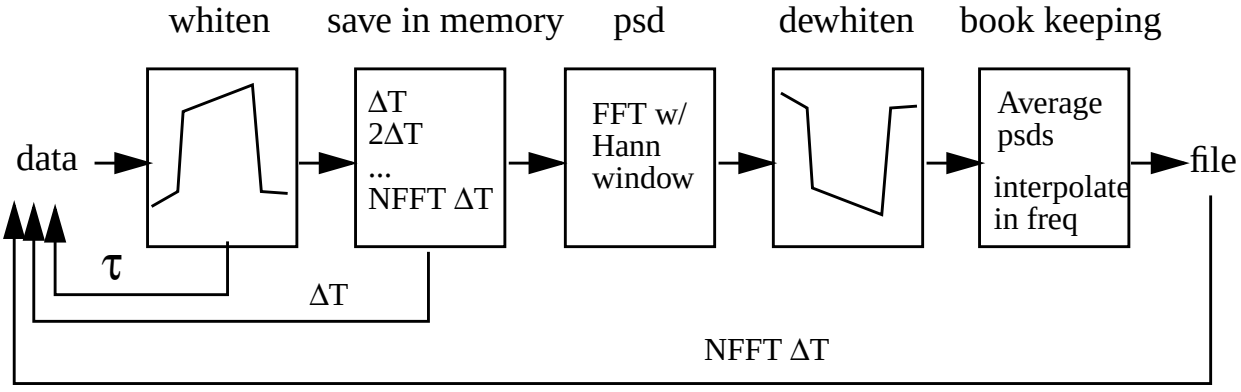


Figure 19: process of psd calculation

The incoming data are stored at every ΔT after applying a bandpass filter to reduce the aliasing and leak from the outside of the frequency window of interest and to whiten the data. After total time of NFFT x ΔT has elapsed, the power spectral density for this cycle is calculated using Hann window to reduce the leak. This is identical to matlab's psd function with the following setting,

psd(val(NFFT data), NFFT, 1/ΔT, hanning(NFFT), NFFT/2) * 2 * ΔT (18)

After the first psd is calculated using NFFT data, the following incoming data are used to calculate successive pdf with NFFT/2 data overlapping. The new psd calculated is dewhitened and the average of psd’s so far calculated is written to the output file.

4.28.2. time step ΔT and duration of simulation NFFT

NFFT and ΔT are calculated internally to optimize the memory and CPU usage. The frequency range is set by f_from and f_to, and power spectral densities at N_freqs frequency points are calculated. If logSpacing is true, frequency points as placed evenly in log scale, and if false, frequencies are placed evenly in linear scale.

The leakage expected for Hanning window is N^{-3}. N is calculated as $(f-f_0)/\Delta f$, where f is the frequency of interest, f_0 is the source of signal, and Δf is the frequency spacing of the FFT calculation.

Chebyshev bandpass filter is applied to reduce the alias effect from both sides, and Lowpass_Order is the order for lowpass filter and Highpass_Order is the one for high pass.

N_Tfft is the minimum number of oscillation of the lowest frequency component, while N_delT is that of the highest frequency component.

The result is stored in a file whose name is the module name with full path prepended. The first row is the list of frequency, and the following lines are averaged psd values. I.e., the first psd line is the result using 1 FFT, second line is the average of 2 FFTs etc. maxCounter is the maximal number of repeats.

This module uses data when the activate port value is non-zero. If one wants to calculate the psd of some output when the system is locked, then set the input “avtive” to 0 until the system is fully locked, and then set the value to 1 after that. Then all data during the lock acquisition process is discarded, and the psd of the in-lock state can be calculated.

real f_from = 0.1

f_from (0.1), f_to (10) real

logSpacing (true) boolean

N_freqs (100), Lowpass_Order (6), Hghpass_Order (6), N_TffT (1), N_delT (4), maxCounter (100) integer

5 MACROS AND SETTINGS BY EXPRESSION

5.1. Macro definition file : e2eDB.mcr

The End to End simulation code supports macros, and one can specify almost all data entries using these macros in stead of typing numerical literals, e.g., “ArmLength/LIGHT_SPEED” or “sin(PI/3)”.

The macros are defined in the following way. When a simulation program starts, it reads in macro definition files named “e2eDB.mcr” in the directories specified in the E2E_PATH environment variable. E.g., when E2E_PATH is“*.:myLibDir:e2eSysDir*”, then the program reads in e2eDB.mcr in “*e2eSysDir*”, in “*myLibDir*” then in the current directory “.”, if there is one. When a macro with the same name is defined in multiple files, the one in the file loaded last is used. With this example

E2E_PATH, the definition in the current directory has the highest precedence and the one in *e2eSysDir* has the lowest.

The format of the macro specification file is as follows. A macro is defined by a line of the following format:

name = value [unit] "comment"

value is a number which will be substituted whenever this macro name is used in the simulation. ***name*** can be composed of any number of alphabets or digits or "_", but the first character cannot be a digit. ***unit*** and ***comment*** are optional strings. Symbols "[", "]" and "" are mandatory if unit or comment is to be defined.

The definition of ***value*** can include macros already defined, and can include mathematical expressions discussed in Section 4.25. A few examples will be in order.

```
elcom = 12.7   [m]   "average of the two lengths"
eldif = 0.3    [m]   "difference of the two lengths"
elIn = elcom + eldif/2   [m]   "inline length"
elOff = elcom - dleif/2   [m]   "offline length"
```

Lines which starts with "%" are treated as comment lines and discarded when reading. In order to printout information about a macro file, like an announcement of a new version, place a line which starts with "<" before the message lines and a line which starts with ">" after the message lines. The message lines can be anything. E.g., if you place a line with "<" at the top of the file and a line with ">" at the bottom of the file, the entire content of the file, except for the two lines, are printed to the console window where the simulation program is started. There can be any number of message groups surrounded by "<" and ">". It will be a good idea to place the following lines at the top of each macro file to clarify what is loaded.

```
<
% database defining H2K IFO parameters
% Updated on September 1 by Hiro Yamamoto
% change : TEMPERATURE is now defined
>
...
...
<
TEMPERATURE = 5/9*72 + 255.37 [K] "global temperature at 72F"
>
```

In addition to these macros defined in external files, there is one set of macros defined internally giving the definitions of various constants, like PI or LIGHT_SPEED.

When you want to see the current macro definition, defined internally and externally, in a given directory, type e2emacro. As of Sep. 1, 2000, the following macros are defined internally.

```
AVOGADRO_NUMBER = 6.0221367e23 "avogadro number"
BOLTZMANN_CONST = 1.38065812e-23 ["m^2 s^-2 kg K^-1"] "Boltzman constant"
LIGHT_SPEED = 2.99792458e8 [m s^-1] "speed of light"
GRAV_ACCEL = 9.80665 [m s^-2] "standard grav. accel. at sea level"
PI = 3.141592653589793
```

Short messages and macro values can be printed using #print direction.

```
#print “This is a messge”
#print LIGHT_SPEED PI
```

This line generates the following output.

```
This is a message
“LIGHT_SPEED” = 2.99792e8
“PI” = 3.14159
```

A macro file can include another macro file by #include directive. An example is

```
#include anotherMacro.mcr
```

IF THEN ELSE conditional controls are supported using #if, #elseif, #else and #endif. All string after these directions (except for #endif) are evaluated as boolean, i.e., false if numerically 0 and true otherwise. The controls can be nexted.

```
LHO2k = 1 % use name instead of numbers
LHO4k = 2
LLO4k = 3
IFO = LHO2k % choose an IFO
#if IFO == LHO2k
CavLength = 2000
#elseif IFO == LHO4k || IFO == LLO4k
CavLength = 4000
#else %something unknown
#print “This IFO is not known” IFO
#endif
```

5.2. Runtime macro specification as a option to the program, -db and -param

You can define extra macros when you run the simulation program.

```
modeler -db myDB.mcr -param mcrname=mcrval -param ‘another=some1*some2’
```

By the run-time option -db, you can force to load a macro definition file. This is loaded all other default files are loaded, so the definitions in this file override all others.

The second option, -param, can be used to define one macro to assigned a value to it. In the above example, a new macro `mcrname` is defined and a value `mcrval` is assigned to it. If a macro of the given name already exists, this definition overrides it. If the right hand size of the assignment has any operators, surround the definition by a pair of single quotes.

These two optioncs can appear multiple times, and the precedence is from right to left of those options.

5.3. Combination of -param and -db

e2eDB.mcr

```
RayleighRange = 1000
```

```
z0 = 2000
dz = 0
#include postproc.mcr
```

postproc.mcr

```
z = z0 + dz
```

```
modeler -param 'dz=RayleighRange*0.01' -db postproc.mcr
```

5.4. Direct macro definition for settings

A macro can be used to define real value settings and / or inputs without touching the primitive itself. The convention is

path1.path2...pathN.instanceName.settingName = value or

path1.path2...pathN.instanceName.inputPortName = value

Path names are optional, but the instanceName is mandatory. This macro capability allows one to set and change settings of primitives without modifying the box files.

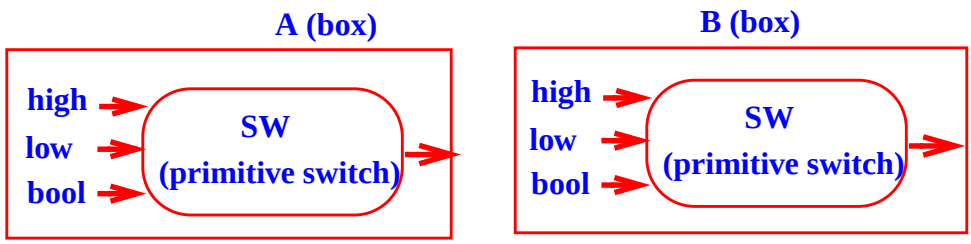


Figure 20: Macro defintion for settings

For example, in Fig. 18, A and B are boxes which contain one instance of switch module named SW. If a macro is defined as

SW.bool = 1

then it is equivalent that the value of bool input of SW in box A and box B are explicitly set to 1. If the definition is

A.SW.bool = 1

then only the bool of SW in box A is set to be 1, and that in box B is not affected.

5.5. Local macro definition

Those macros defined in macro files apply to all box files. Local macros can be defined which apply only to those primitives and boxes contained in a box where these macro are defined. For example, a macro named TEMPERATURE is defined as a global variable. If you want to assign a different tempretature to a subsystem, keep the subsystem in a box, say susSys.box, and define macro TEMPERATURE with the subsystem temperature in the box susSys.box. Local macros can be define in a box using alfi GUI program.

Another example is a Fabry Peroit cavity box. The two propagators should use the same length for the propagation length. In order to do that, you define a macro FPlength in the box and you use the this name in the settings of the two propagators.

5.6. Examples

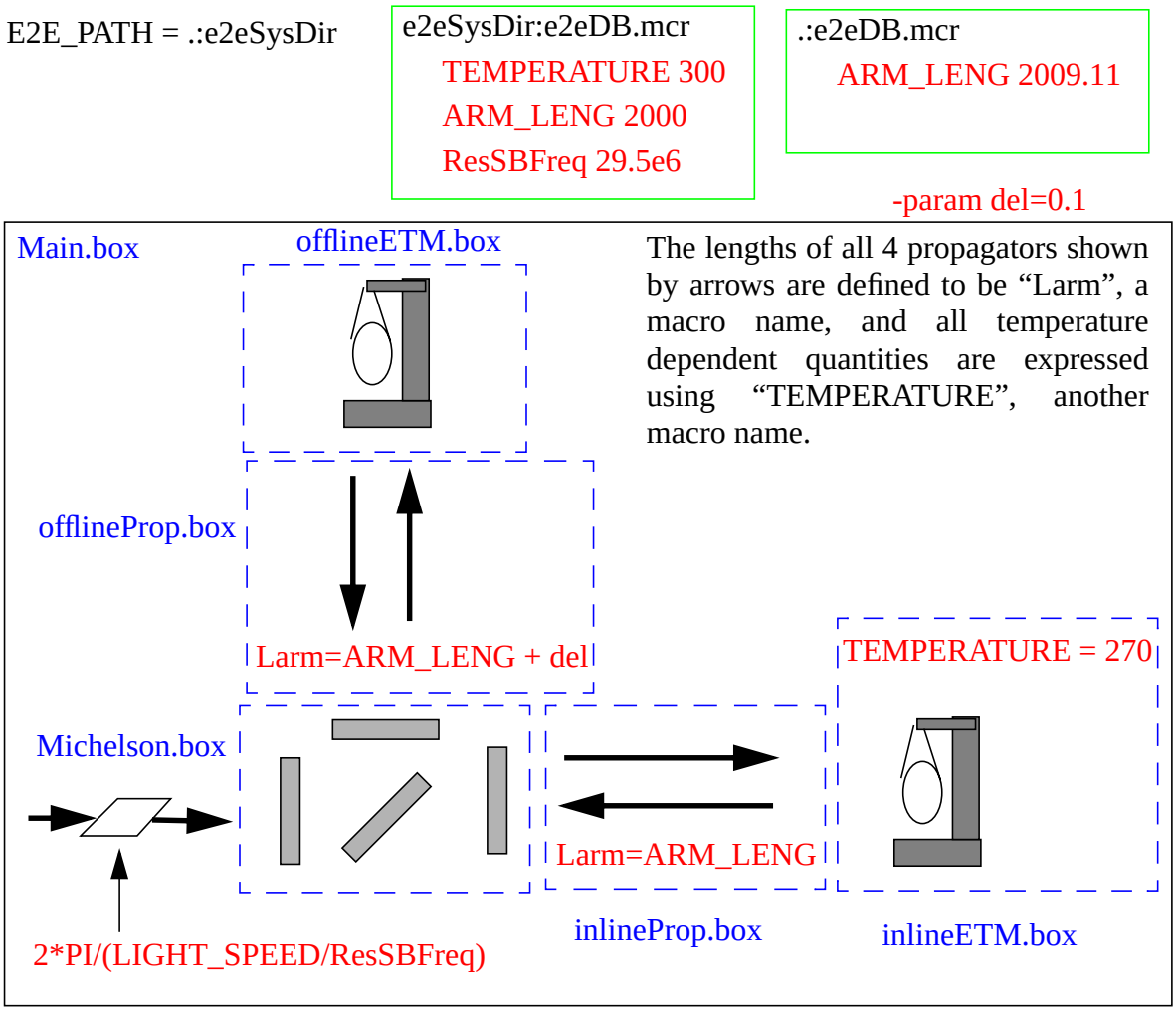


Figure 21: macro definition example

In Fig. 19, there are two e2eDB.mcr files. When you run modeler as

```
modeler -param del=0.1
```

there are four macro names defined, TEMPERATURE, ARM_LENG, ResSBFreq and del, whose values are 300, 2009.11, 29.5e6 and 0.1 respectively. The definition of ARM_LENG in the current directory overrides the definition in the e2eDB.mcr file in e2eSysDir directory.

In inlineETM.box, the macro TEMPERATURE is defined as 270. Any settings using TEMPERATURE in inlineETM.box use 270 for the TEMPERATURE, while all others use 300. The arm length of the inline cavity is 2009.11, while that of the offline cavity is 2009.21.

The phase modulation is specified by k, the wave number. This setting can be expressed using the modulation frequency ResSBFreq and predefined constants PI and LIGHT_SPEED.

6 RUN TIME OPTIONS OF THE SIMUALTION PROGRAM

6.1. -bin

The simulation program stores the output in a binary file, in stead of an ascii file, which is the default. In matlab, a binary file can be loaded by using e2ebin.m. The format is

```
[vals, titles] = e2ebin('binary file name');
```

The binary file can be converted to an ascii file with an associated header file by using e2ebinLoader (see Sec. 7.4.).

6.2. -d1, -d2, -d3, -d4

The simulation program prints auxiliary information. -d4 provides most comprehensive information while -d1 the least.

6.3. -db [dababase file], -param name=vale (see Section 6)

With -db, you can specify a macro file to be loaded after all the default macro dabatase are loaded. One specific macro can be specified by -param option. If some operators are in the right hand size of the parameter definition, surround the definition by a pair of single quotes. The format for these parameters are

```
modeler -db myDB.mcr -param del=0.1 -param 'val=val1*val2'
```

6.4. -help

Explains these runtime options.

6.5. -prof [output file name] [number of modules reported]

The time spend used in each module are analyzed. The output file can be specified where the profiling information is stored. In the profiler output, top 50 modules are reported. If a number is passed as a part of -prof option, you can change the number of modules reprted.

6.6. -seed seedVal

The seed for the random number generator can be specified. If this is not specified, data/time is used to generate a seed.

6.7. -v, -V

Print version number.

6.8. -maxiter=number (for modeler_freq)

The maximal number of iteration tried in modeler_freq to calculate the transfer function. The default is 500. If modeler_freq generates messages that “convergence failed”, try to give a larger value, like 5000.

7 E2E AUXILIARY PROGRAMS

7.1. detmap

The pd_demod (Section 4.16.) module simulating the photo detector and demodulation process uses a data file to calculate the response of a detector with an arbitrary shape. detmap generates this data file for a wide range of detector shapes. The detector shape can be defined interactively in this program, and the nonuniformity of the detector surface can be specified as well.

7.2. e2emacro

Prints the current macro settings.

7.3. e2ecalc

e2e calculator

7.4. e2ebinLoader

Converts the binary format output file to an ascii file. The format is

```
e2ebinLoader [-d] [binaryFileName] [-help]
```

With -d option provided, this program prints auxiliary information during decoding. The ascii data file created has a file extension .asc, while the data header file (see Sec. 8.5.) has .dhr.

8 FILE TYPES

8.1. .box (edited by alfi, input to simulation program)

The box is created by alfi and contains the definition of the setup to be simulated. The syntax of this file is defined in Section 10.

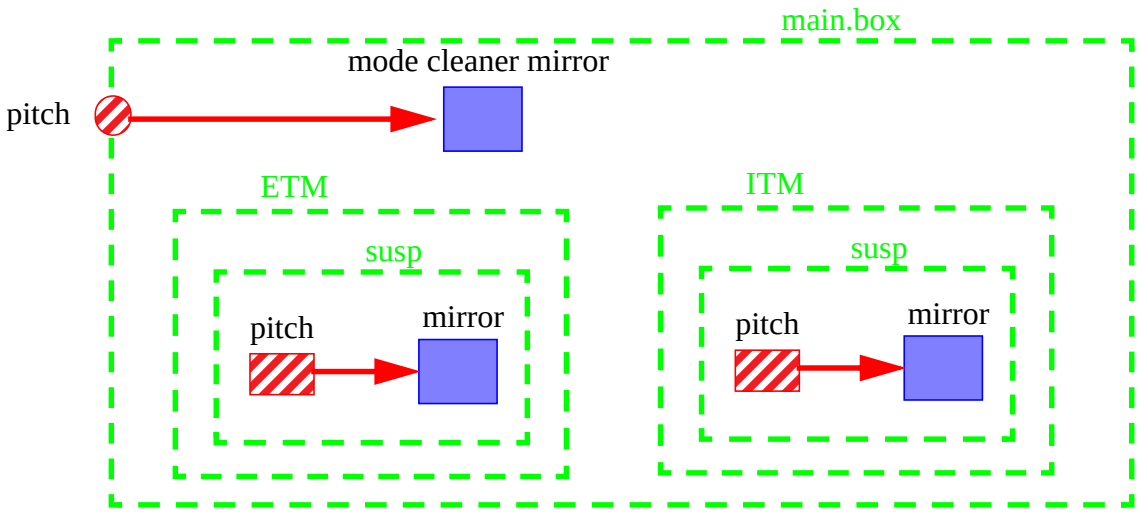


Figure 22: .par file settings

8.2. .par (edited by text editor, input to simulation program)

In a parameter file, values can be assigned to input ports of the outmost box or to data_in modules anywhere contained in the simulation definition. You specify which .par file to use when you run the simulation program.

The difference between macro and parameter is that values of parameters can be changed during a run, while macro values cannot be. When you run a time domain simulation, you specify how long the simulation should run. After that amount of data have been simulated, you are prompted if you want to continue. If you request to keep simulating, the program reads the .par file again and keep running for another amount of time you specified. If you modify the .par file before you resume the simulation, that new value is used in the second part of the simulation. E.g., you can change a gain value.

In a .par file, a line which starts with % is a comment line.

In Fig. 20, there are 5 boxes, main, the out most one, ETM, ITM and susp contained in ETM and ITM. The circle named pitch is an input port name of the main box, while two boxes named pitch are data_in modules. In the .par file, the following specifications are possible.

```
% next applies to all 3
pitch = 1e-6
% next applies to 2 data_ins
susp.pitch = 1e-6
% next applies to only the pitch in ETM
ETM.susp.pitch = 1e-6
% next applies to only the pitch in ITM
ITM.susp.pitch = 1e-6
```

When you specify only the name, all input ports of the main box and all data_in modules with that name are assigned this specification. If the specification of a data_in name contains box names

containing that module, this specification applies to only those data_ins which satisfy that box hierarchy.

The is a token, #TIME, to automate the change of parameter settings. The syntax is

```
lines0
#TIME t1
lines1
#TIME t2
lines2
...
#TIME tM
linesM
```

where linesN are legal lines of .par files, like pitch = 0, and tN is the time since the start of the simulation run. When a simulation job runs for a period of T with this .par file, the simulation uses definitions by lines0 for the first time period t1, then uses definitions lines0 and lines1 for the following period of (t2-t1), and so on.

A simple example will clarify the point.

```
pitch = 0
lock = 0
#TIME 0.5
lock = 1
#TIME 1
picth = 1e-8
```

When you run a job for 5 seconds, the simulation uses pitch=0, lock=0 for the first 0.5 seconds, then sets lock=1 and runs for another 0.5 seconds. After running total of 1 second, the pitch value is changed to 1e-8. This can be easily understood when you remember the rule that the definition given later has a higher precedence than the one given before.

The run time does not need to be longer than the time specified for the last #TIME token. In the above example, if you run only for 0.7 seconds, simply the pitch value will remain 0 for the entire run.

8.3. .in (input to simulation program)

When you run the simulation program, the key strokes can be stored in a file so that you can easily run the program again. The name can be anything, but .in is a popular extension used. At any prompt, when you type

```
@(filename
```

a file named “filename” is created and all prompts and your key entries are stored in the file, until you type

```
@)
```

or until the end of the run.

When you modify the box files, and want to rerun again, you run the program and type

```
@filename
```

then, all the recorded key strokes are replayed. You can use this file as an input stream to the program, like

```
cat filename - | modeler
```

or

```
modeler < filename
```

The input sequence file is a text file and you can edit it, e.g., change the simulation time or time step. A line which starts with "%" is a comment line and you can add your own.

If you add a line

```
@PROMPTOFF
```

then, at the replay time, the prompts and replies will not be printed on the console window until the line

```
@PROMPTON
```

8.4. .dat (output of simulation program)

The output file. The first column is the time or frequency, and the rest columns are data specified by the output ports of the out most boxes or data_out modules. The arrangements of those columns are given in the associated .dhr file (see Sec. 8.5.).

8.5. .dhr (input to and output of simulation program)

The data header file contains the titles of data in the associated .dat file. E.g., lock.dat has 4 columns of data series, time, power, inDemod, outDemod. Then lock.dhr is like the following.

```
time
power
inDemod
outDemod
```

When the simulation program runs and tries to create the output data file, it looks for the associated .dhr file. If there is one, then the order of the columns follows the order specified in the .dhr file. E.g., if there were lock.dhr file whose content is

```
time
outDemod
inDemod
power
```

then, the data column in lock.dat is arranged following this order, i.e., second column is outDemod and power is placed in the 4th column.

Time and frequency should be always the first one. If there is no .dhr file, the order of data are decided following an internal rule.

8.6. .set (output of simulation program)

A setting file is created when a simulation program runs. It contains a list of all settings of all modules used in that simulation, along with other data, like the random number seed and the full macro definition.

8.7. .prm, .xbm (input to alfi)

This file contains all the interface information, what are the names and types of inputs and outputs etc, of a primitive with the same nane, i.e., LocAcq.prm is for a primitive module LocAcq. These .prm files are used by alfi. Alfi reads these files and finds how to display each primitive (number of ports and name, etc). .xbm is a bitmap file which contains an icon for a primitive represented by a .prm 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 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

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 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:

$$\phi(t) = \int_0^t \omega(t)dt = \omega_0 \cdot t + \int_0^t \delta\omega(t) \qquad (19)$$

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 DESCRIPTION FILE SYNTAX

10.1. 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.

10.2. Syntax

Blank lines can be inserted.

% all the rest after % is treated as comments.

Add_Macros
{
name1 = val1
...
nameN = valN
}

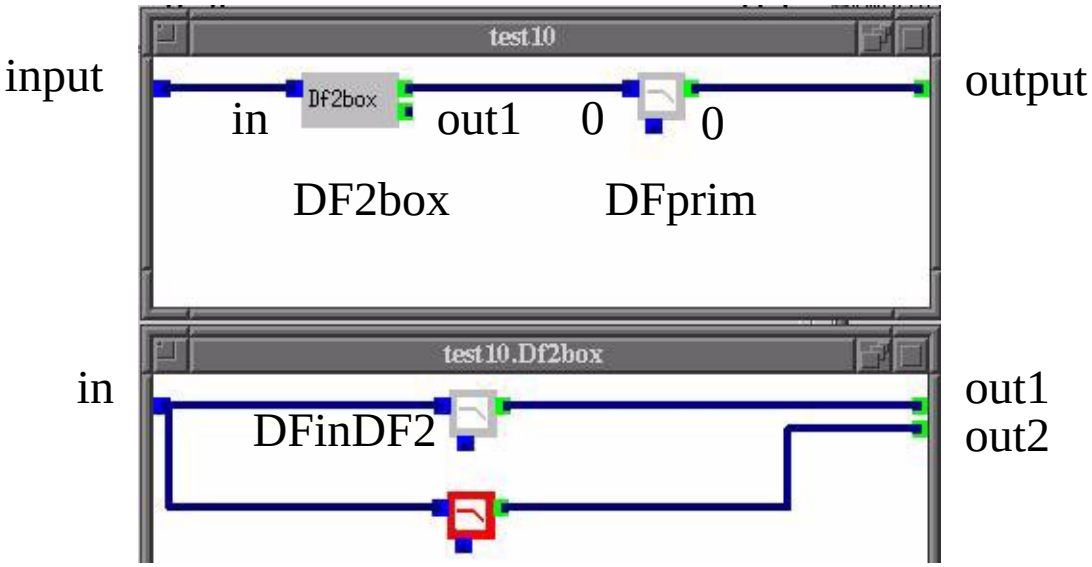
Add_Submodules

```
{
    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
    ...
}
```

10.3. Example




```
Add_Macros
{
  gainVal = 2*PI
}

Add_Submodules
{
  box DF2box { #include DF2.box }
  digital_filter DFprim
}
Settings DFprim
{
  pole = -1
}
Settings DF2box
{
  Settings DFinDF2
  {
    gain = gainVal
  }
}
```

```
Add_Connections
{
  this input -> DF2box in
  DF2box out1 -> DFprim 0
  DFprim 0 -> this output
}
```

10.4. Explanation of the syntax

10.4.1. Add_Macros

Add_Macros section surrounded by { and } defines macros effective in this box.

10.4.2. 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.

10.4.3. 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.

10.4.4. 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".

10.5. 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”

[5] M. Rakhmanov. “Dynamics of Laser Interferometric Gravitational Wave Detectors” PhD thesis, California Institute of Technology, Pasadena, California, 2000.

[6] S. Kawamura and J. Hazel. LIGO-T970135 “Small Optics Suspension Final Design (Mechanical System)”
S. Kawamura, J. Hazel and M. Barton. LIGO-T970158 “Large Optics Suspension Final Design (Mechanical System)