NAME

shapes – input file for the nanophotonic simulation program **shapes**.

SYNOPSIS

Parallel version: **mpirun** –np *number-of-processes* –machinefile *node-file* **shapes** *shapes-input-file*

Sequential version: **shapes** *shapes-input-file*

DESCRIPTION

Program **shapes**(1) takes input from a simple configuration file that describes the modeled system. This document describes how to construct this file.

USAGE

The input file to **shapes**(1) is a simple text file with various keywords, one per line, and their values separated by an equal sign. Lines that begin with the hash character are comments that are neglected by **shapes.**

The keywords may appear in any order, but it is a good practice to organize them in logical groups. And so the following groups are used by the current version of **shapes:**

chat	*signal*
watch	*metal*
level0	*tag*
iterate	*refine*
pml	*spectral*
output	

In the text below entries that are flagged as *int* must be entered as integers. Entries that are marked as *Real* may be entered as reals or as integers. Arrays must be input as a row of numbers separated by spaces, all typed on a single line.

The *chat* group

There are 8 keywords in this group currently.

chat.print_versions int
: When set to 1, the program prints RCS versions of all component files. Setting it to 0 disables this output.

chat.chombo_verbose int
: Activates chatting by C++ functions when set to 1 or 2. When set to anything higher than 2 it may trigger Fortran messages too. Disables the chat when set to 0.

chat.fortran_verbose int
: Activates chatting by Fortran subroutines when set to 1 or higher. Disables the chat when set to 0.

chat.print_level_0_domain int
: Once the level 0 has been constructed **shapes** prints its geometry and other level 0 related parameters when **chat.print_level_0_domain** is set to 1 or higher. Setting it to 0 disables this output.

chat.print_levels int
: When this parameter is set to 1 **shapes** prints information about every level that has been constructed. Setting this parameter to 1 sets **chat.print_level_0_domain** to 1 too.

chat.print_min_max int
: When this parameter is set to 1 or higher **shapes** prints minimum and maximum values of all fields for all levels for the time slice at which data is dumped. Historical minimum and maximum values for the fields are printed too. Setting this parameter to 0 disables this output.

chat.print_actions int
: Setting this parameter to 1 makes **shapes'** functions (the Chombo shell ones) print all they do. This can be useful in deciphering recursion, checking synchronization between levels and making sure that data flows between the levels at appropriate times and in the correct direction. This parameter is much the same in practice as **chat.chombo_verbose**.

chat.print_dots int

When this parameter is set to 1 or higher, **shapes** prints a dot for every iteration. It may be useful to set it for interactive sequential runs, so that the user knows that something happens and how much work has been done so far. Whenever other diagnostic output is enabled, it is best to switch it off.

Example

To make the program shut up altogether and just print a dot for each iteration set

```
chat.chombo_verbose             =    0
chat.fortran_verbose            =    0
chat.print_levels               =    0
chat.print_versions             =    0
chat.print_min_max              =    0
chat.print_actions              =    0
chat.print_dots                 =    1
```

When the program runs in parallel, every participating MPI process writes its own diagnostic output, if requested, on its own file called *pout.X* where *X* evaluates to the rank number of the process.

The *watch* **group**

This group lets a user and even more so a programmer watch specific functions and subroutines in action. There are some 40 C++ functions in the program, each of which can be selectively asked to talk, while the others remain mute. This way of tracing the actions of the program is much more effective in pin-pointing eventual problems than requesting a blanket *chat.chombo_verbose* or *chat.print_actions*. If a given C++ function calls a Fortran subroutine, then setting its *watch* parameter to an integer larger than 2 activates *chat.fortran_verbose* for this subroutine, but not for the other ones.

The way to make a select C++ function talk is to set *watch* followed by a dot and the name of the function to an integer larger than 0. For example to make function *push_d* talk set

watch.push_d = 1

The following table lists the names of the C++ **shapes** functions that can be watched thusly.

advance_e	*exchange_media_fields*
advance_h	*full_copy*
analyze_levels	*initialize_level_data*
build_basic_fields	*inject_d*
build_cell_centered_fields	*inject_h*
build_distributions	*interpolate_basic_fields*
build_disjoint_box_lyout	*interpolate_fourier_fields*
build_fourier_fields	*interpolate_media_fields*
build_level	*main*
build_media_fields	*patch_basic_fields*
build_minmax	*patch_fourier_fields*
convert_d_to_e	*patch_media_fields*
copy_basic_fields	*push_d*
copy_fourier_fields	*push_h*
copy_media_fields	*regrid*
dump_data	*tag_cells*
effective_domain_size	*time_interpolate*
evaluate_energy	*write_box_layout_data*
exchange_basic_fields	*write_gnuplot_data*
exchange_fourier_fields	*write_tags_data*

Example

To see how functions *convert_d_to_e*, *push_d*, and *push_h* go about their business and to trigger diagnostics in Fortran subroutines invoked from *convert_d_to_e* set

```
watch.convert_d_to_e            =    3
watch.push_d                    =    1
watch.push_h                    =    1
```

The *level0* group

This group is used to specify the general geometry of the computational domain. The computational domain is always rectangular. We have to specify its length and width, as well as the number of grid divisions in each direction. Additionally we have to specify how the computational domain is going to be distributed amongst MPI processes if the program is going to run in parallel.

The geometry of the computational domain is specified in arbitrary units of length. For example, suppose we are going to inject a plane harmonic wave of length 5000 Angstroms into the computational domain, which is going to be a square. We would like to resolve the wave on, say, 40 grid segments and we would like to fit up to four full wavelengths into the total field region.

This can be done as follows. Let us make the computational region 100×100 units and let us make the grid divisions in both directions $\Delta x = \Delta y = 0.5$. The unit of length is therefore going to be $2 \times \Delta x = 1$ (which is why it is called a *unit*).

This choice of Δx and Δy results in a 200×200 grid. Now we want to resolve our 5000 Angstrom long wave on 40 grid segments, which means that each $\Delta x = \Delta y = 0.5 = 5000\text{A}/40 = 125\text{A}$. The unit of length is therefore going to be $2 \times \Delta x = 250\text{A}$ and the wavelength itself is $40 \times \Delta x = 40 \times 0.5 = 20$ *our* units of length. In order to fit four full wavelength in the total field region in each direction the region must have the length and width of $20 \times 4 = 80$ units. If we were to place the region centrally within the computational domain of 100×100 units, its corners would be $(10, 10)$ and $(90, 90)$.

shapes can perform its computations on a multi-grid. Here we define only the so called *level 0* grid, i.e., the *coarsest* grid. Higher level grids, i.e., finer grids, are built by the program automatically. The user can specify where finer gridding should be used, if at all, by using keywords of the *tag* group. It is *not* necessary to use the multi-grid though and it is generally better not to.

We have 8 keywords in the *level0* group with which we can convey all the information we have evaluated above to **shapes.**

level0.nx int
: Number of level 0 cells in the x direction.

level0.ny int
: Number of level 0 cells in the y direction.

level0.nbx int
: Number of boxes in the x direction. **shapes** will group the level 0 grid cells into boxes and will distribute them amongst MPI processes if executed in parallel. **shapes** will try to make the boxes square, so the number of boxes in the x direction defines the average size of a box *both* in the x and in the y directions. If **shapes** is run sequentially then it is best to set this number to 1. But it is OK to set it to something higher, e.g., 2 or 4. In this case **shapes** will group all level 0 grid cells into separate boxes and will go through all the motions of moving ghost data between the boxes, as if it was running on multiple CPUs. But all these operations will take place within memory of the single sequential process. This can be used, for example, in test runs before submitting a job to a multi-computer.

level0.x0 Real
: Value of the x coordinate of the $(0, 0)$ point of the level 0 grid.

level0.y0 Real
: Value of the y coordinate of the $(0, 0)$ point of the level 0 grid.

level0.delta_x Real
: Level 0 grid constant in the x direction, i.e., Δx.

level0.delta_y Real

Level 0 grid constant in the y direction, i.e., Δy.

level0.time Real

The value of initial time that corresponds to the $\vec{E}$ field within the level 0 grid. The $\vec{H}$ field is defined on a time slice that is $\Delta t/2$ (Δt for level 0 is defined in the iteration group) ahead of the $\vec{E}$ time slice.

Example

The configuration discussed in this section would be described as follows with an additional specifications in the *signal* group:

```
level0.nx                       =  200
level0.ny                       =  200
level0.nbx                      =    1
level0.x0                       =    0
level0.y0                       =    0
level0.delta_x                  =    0.5
level0.delta_y                  =    0.5
level0.time                     =    0
   ...
signal.x_lo                     =   10
signal.y_lo                     =   10
signal.x_hi                     =   90
signal.y_hi                     =   90
signal.lambda                   =   20
```

The *iterate* **group**

This group tells **shapes** how the level 0 should be iterated, i.e., what should be the iteration time step, how many iterations should be performed altogether and how often should field images by dumped.

shapes performs its computations in the natural units in which the speed of light in vacuum, c, is 1. This means that in one unit of time the wave front is going to propagate by one unit of length in vacuum. If we were to re-use the example discussed in the previous section, where the unit of length has turned out to be 250 Angstroms we would end up with the unit of time of

$$250\text{A}/c = 250\times 10^{-10}\text{m}/2.997925\times 10^{8}\text{m/s} = 8.3391\times 10^{-17}\text{s}$$

This number will be needed to convert various material properties to natural units, but it is not needed to define iteration parameters of this section.

Instead, the iteration time step is specified by telling **shapes** in how many steps is the wave front going to traverse a single grid cell in the x direction, i.e., how many iterations are needed to push the wavefront through a distance of Δx. This number is called *level 0 stride.* And so, if we set, say, level 0 stride to 4, the time step will be chosen so that Δx (in the case of the example discussed above, this was 250 Angstroms) will be traversed in 4 level 0 iterations.

It is not always the best idea to make the time step as long as the stability criterion for vacuum electrodynamics lets it be, i.e., $\Delta t = \Delta x/\sqrt{n}$ where n is the number of dimensions. The reason for this is that this number does not take into account stability conditions for the auxiliary differential equations. In the presence of a multi-grid, this number may have to be shortened too.

There are 4 keywords in this group

iterate.level0.stride Real

This is the level 0 *stride* discussed above.

iterate.level0.image_frequency int

This parameter tells **shapes** how often to dump images. **shapes** will dump images every *iterate.level0.stride* times *iterate.level0.image_frequency* time steps. For example, if *iterate.level0.stride* is 4 and *iterate.level0.image_frequency* is 2, the image will be dumped every

$4 \times 2 = 8$ time steps. Because in 4 time steps the wave front propagates by one Δx the wave front will be shifted by $2 \times \Delta x$ between adjacent images. In the case of the example discussed above, the wave front shift between adjacent images will be by one unit of length.

iterate.level0.number_of_steps int
This number tells **shapes** how many time steps to perform altogether.

iterate.use_substep int
This parameter matters only when the computations are carried out on a multi-grid. If it is set to 0 **shapes** will use the same time step for each multi-grid level. This, of course, implies that in this case the time step should be chosen based on the finest grid level spacing requested. If it is set to 1 **shapes** will use a shorter time step for finer grid levels. The space refinement ratio between adjacent grid levels is 2 and the time refinement level between adjacent grid levels is 3. This is required to synchronize $\vec{E}$ and $\vec{H}$ time slices between adjacent levels.

Example

```
iterate.level0.stride            =    4
iterate.level0.image_frequency   =    4
iterate.level0.number_of_steps   = 4800
iterate.use_substep              =    0
```

Here we are going to time-step so that the wave-front crosses a single level 0 Δx in 4 time steps. Images will be dumped every $4 \times 4 = 16$ level 0 time steps and the total number of level 0 time steps will be 4,800. The wave front will traverse $1200 \times \Delta x$ in this time. If $\Delta x = 0.5$ units of length, as in the above example, the total distance travelled by the wave front throughout the simulation will be 600 units of length.

The *pml* group

PMLs are characterized just by specifying the boundary of the PML region. The user needs to specify the lower left and the upper right corners of the boundary box. **shapes** takes care of all the rest. The keywords used to do this are

pml.x_lo Real

pml.y_lo Real
The x and y coordinates of the lower left corner of the PML boundary box.

pml.x_hi Real

pml.y_hi Real
The x and y coordinates of the upper right corner of the PML boundary box.

Example

Consider again the example discussed above. Here we have decided on the total field region being restricted to a box defined by the $(10,10)$ and $(90,90)$ corner points. We can therefore define our PML region by the $(5,5)$ and $(95,95)$ corner points. This is still within the $(0,0)$ and $(100,100)$ computational domain and it yields 10 grid segments to attenuate the signal on each boundary. The distance between the PML boundary and the total field region boundary is 5, which is again 10 grid segments:

```
pml.x_lo                         =  5
pml.y_lo                         =  5
pml.x_hi                         = 95
pml.y_hi                         = 95
```

The *signal* group

shapes defines 11 signals, which are summed up in the following table.

mode	description	formula
0	nothing	
1	harmonic wave	$f(\zeta) = \sin\left(\frac{2\pi}{\lambda}\zeta\right)$
2	step ramped harmonic wave	$f(\zeta) = \theta(-\zeta)\sin\left(\frac{2\pi}{\lambda}\zeta\right)$
3	tanh ramped harmonic wave	$f(\zeta) = \frac{1}{2}(1-\tanh(\alpha\zeta))\sin\left(\frac{2\pi}{\lambda}\zeta\right)$
4	Gaussian pulse	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)$
5	Gaussian envelope harmonic wave	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda}\zeta\right)$
6	Gaussian envelope linear chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\beta\zeta}\zeta\right)$
7	Gaussian envelope quadratic chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\beta\zeta^2}\zeta\right)$
8	Gaussian envelope exp chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\alpha\exp(\beta\zeta)}\zeta\right)$
9	Gaussian envelope sin chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\alpha\sin(\beta\zeta)}\zeta\right)$
10	Gaussian envelope tanh chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\alpha\tanh(\beta\zeta)}\zeta\right)$
11	Gaussian envelope Gaussian chirp	$f(\zeta) = \exp\left(-\frac{\zeta^2}{2\sigma^2}\right)\sin\left(\frac{2\pi}{\lambda+\alpha\exp\left(-\frac{\zeta^2}{2\beta^2}\right)}\zeta\right)$

The various chirps work by modulating the wavelength of the basic signal either linearly or in some other way. Normally we want this modulation to be slow compared to the fundamental wavelength λ and to the length of the envelope. Otherwise we'll get a complicated beat instead of a chirp. It is normally best to set the parameters of a chirp or any other signal by drawing it with, e.g., gnuplot, before entering them into the **shapes** input file. One should especially ensure that the effective wavelength is not going to become zero during the program execution. This may happen when using a linear chirp. But a tanh chirp is a good alternative here, because we can set a minimum and a maximum wavelength in this model and then modulate the effective wavelength smoothly and almost linearly in between.

The Gnuplot command to plot a function such as one of the above is

```
set sample 800,800
plot [z=-200:200] [-1.5:1.5] \
   f(z) = 0.5 * (1 - tanh(a * z)) * sin(6.2831853 * z / l), \
   a = 0.02, l = 20, f(z)
```

Using this command the user can play with a (i.e., α) and l (i.e., λ) until the right shape of the pulse is found.

The signal shape defined by a given mode and its various constants is injected into the computational region by substituting

$$\zeta = n_x(x - x_0) + n_y(y - y_0) - (t - t_0)$$

The actual fields that are injected into the total field region are then

$$H_z = f(\zeta)$$
$$E_x = -n_y f(\zeta)$$
$$E_y = n_x f(\zeta)$$

One can easily check that these indeed satisfy the Maxwell vacuum equations.

The parameters x_0 and y_0 are set to either the lower or the upper value of x and y respectively within the total field region, depending on the direction from which the signal is injected. But t_0 is set by the user. This parameter specifies how far from the total field region boundary is the center of the signal when the program begins its execution.

The input file parameters that define the signal are now as follows.

signal.x_lo Real

signal.y_lo Real
The x and y coordinates of the lower-left corner of the total field region.

signal.x_hi Real

signal.y_hi Real
The x and y coordinates of the upper-right corner of the total field region.

signal.mode int
A type of signal to be injected, see the table above.

signal.t0 Real
Signal delay.

signal.lambda Real
The fundamental wavelength λ - it may get modulated in chirps.

signal.sigma Real
The half-width of the Gaussian pulse or envelope.

signal.alpha Real

signal.beta Real
Additional parameters to specify the signal, see the table above.

signal.vx Real

signal.vy Real
The x and y components of the signal direction vector. They don't have to be normalized. **shapes** uses v_x and v_y to calculate n_x and n_y.

Example

```
signal.x_lo                    =  10
signal.y_lo                    =  10
signal.x_hi                    =  90
signal.y_hi                    =  90
signal.mode                    =   7
signal.t0                      = 300
signal.lambda                  =  10
signal.sigma                   =  60
signal.alpha                   =   0
signal.beta                    =   0.0012
signal.vx                      =   0
signal.vy                      =   1
```

Here we define our total field region by specifying two corners $(10,10)$ and $(90,90)$ within the computational domain. The signal that is going to be injected will be a Gaussian envelope quadratic chirp. The signal is going to be injected with the delay of 300, i.e., 300 time units will have to pass before the signal center enters the total field region. The fundamental wavelength λ is 10. In this case, because the chirp is quadratic and β is positive, the signal will start with a longer wavelength, then the wavelength will get progressively shorter until it reaches $\lambda = 10$ and then it will stretch again. The half-width of the Gaussian envelope is 60 and the β parameter is 0.0012. The signal will propagate in the y direction.

The *metal* group

This group is used to specify the media and their layout. The user can specify an arbitrary number of media that can be distributed in a quite arbitrary way within the total field region of the computational domain.

The media models is a multiple resonance Drude/Lorentz model described by the following equation

$$\vec{D}(\omega) = \left(\varepsilon_\infty + \sum_k \frac{\varepsilon_k}{\alpha_k + i2\delta_k(\omega/\omega_k) - (\omega/\omega_k)^2} \right) \vec{E}(\omega)$$

Observe that ε_0 is absorbed into $\vec{D}$. We also absorb μ_0 into $\vec{H}$ so that these two constants do not float around the code. Consequently, in vacuum we have that $\vec{E} = \vec{D}$.

The above formula captures both Drude and Lorentz models. For a Drude model simply substitute

$$\begin{aligned} \omega_k &= \omega_D \\ \alpha_k &= 0 \\ \delta_k &= \frac{\Gamma_D}{2\omega_D} \\ \varepsilon_k &= 1 \end{aligned}$$

which yields a Drude term

$$-\frac{\omega_D^2}{\omega^2 - i\Gamma_D\omega}$$

The *metal* group is internally divided into various subgroups of which the first one is the *metal.media* subgroup. It is here that we define the actual metals. The definition begins with two parameters.

metal.media.number_of_media int
: This parameter tells **shapes** how many different metals we are going to have in the system.

metal.media.number_of_terms int
: This parameter tells **shapes** how many resonance terms we are going to use for each metal. All metals must be defined by the same number of resonances. If we have two metals and one is defined by 3 resonances, whereas the other one by 2 resonances only, then the second metal must be still entered in therms of 3 resonances, but the last resonance should be set to zero.

The numbers that characterize the metals must now be entered as a vectors of reals, i.e.,

metal.media.epsilon_infty vector of Real
: The values of ε_∞ for each of the metals must be entered in a single line and separated by spaces.

metal.media.omega vector of Real
: The values of ω_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals enter first the numbers for the first metal, then for the second metal, and so on.

metal.media.alpha vector of Real
: The values of α_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals enter first the numbers for the first metal, then for the second metal, and so on.

metal.media.delta vector of Real
: The values of δ_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals enter first the numbers for the first metal, then for the second metal, and so on.

metal.media.epsilon vector of Real
: The values of ε_k for each of the metals must be entered in a single line and separated by spaces. For multiple resonance metals enter first the numbers for the first metal, then for the second metal, and so on.

Example

```
metal.media.number_of_media = 2
metal.media.number_of_terms = 3
#-------------------------------------------------------------------------
#                            metal 1                   metal 2 (pure Drude)
#-------------------------------------------------------------------------
#                            k = 1    k = 2   k = 3     k = 1   k = 2   k = 3
#-------------------------------------------------------------------------
metal.media.epsilon_infty   = 2.36461                   2.36461
metal.media.omega           = 0.66285  0.33229 0.39318   0.66285 0       0
metal.media.alpha           = 0        1       1         0       0       0
metal.media.delta           = 0.00429  0.06393 0.10576   0.00429 0       0
metal.media.epsilon         = 1        0.31504 0.86805   1       0       0
#-------------------------------------------------------------------------
```

The actual values must be given in natural units. This is not hard to do because ε_∞, ε_k, α_k and δ_k are all unit-less. So they can be evaluated in any units and whatever comes out can be typed into the **shapes** input file right away. The only quantities that have to be converted are the omegas. And here the rule is as follows. I demonstrated how to evaluate the length of the natural unit of time in seconds above. Let us call this number Δt. Let us call a frequency expressed in terms of the natural unit of time $\omega_{\Delta t}$ and frequency expressed in Herz (1/s) ω_s. Then

$$\omega_{\Delta t} = \omega_s \frac{1}{\mathrm{s}} = \omega_s \frac{1}{\mathrm{s}} \left(\frac{\Delta t}{\Delta t} \right) = \omega_s \left(\frac{\Delta t}{\mathrm{s}} \right) \frac{1}{\Delta t}$$

In other words, all that needs to be done is to multiply ω_s by Δt/s, which is the length of the natural unit of time in seconds.

The next subgroup of parameters, *metal.mask*, tells **shapes** where we *do not* want to have any metal. It can be useful sometimes to define media layout in terms of masks. A mask is defined by a combination of basic shapes the program knows how to handle. These are rectangles (here for historic reasons called *boxes*) triangles and circles (here for historic reasons called *cylinders*). The user may define any number of these including none. Wherever a mask figure is declared **shapes** will *not put any metal there. The figures may overlap.*

The keywords in this subgroup are as follows.

metal.mask.boxes.number int
: Number of mask boxes that are going to be defined.

metal.mask.boxes.x_lo vector of Real

metal.mask.boxes.y_lo vector of Real
: x and y coordinates of the lower left corners of the boxes. Each vector must contain *metal.mask.boxes.number* of elements.

metal.mask.boxes.x_hi vector of Real

metal.mask.boxes.y_hi vector of Real
: x and y coordinates of the upper right corners of the boxes. Each vector must contain *metal.mask.boxes.number* of elements.

metal.mask.cylinders.number int
: Number of mask cylinders that are going to be defined.

metal.mask.cylinders.xc vector of Real

metal.mask.cylinders.yc vector of Real
: x and y coordinates of the centers of the cylinders. Each vector must contain *metal.mask.cylinders.number* of elements.

metal.mask.cylinders.rc vector of Real
: Radii of the cylinders. Each vector must contain *metal.mask.cylinders.number* of elements.

metal.mask.rings.number int
Number of mask rings that are going to be defined.

metal.mask.rings.xc vector of Real

metal.mask.rings.yc vector of Real
x and y coordinates of the centers of the rings. Each vector must contain *metal.mask.rings.number* of elements.

metal.mask.rings.r_lo vector of Real

metal.mask.rings.r_hi vector of Real
Low and high radii of the rings. Each vector must contain *metal.mask.rings.number* of elements.

metal.mask.ellipses.number int
Number of mask ellipses that are going to be defined.

metal.mask.ellipses.xa vector of Real

metal.mask.ellipses.ya vector of Real
x and y coordinates of the first focus of the ellipses. Each vector must contain *metal.mask.ellipses.number* of elements.

metal.mask.ellipses.xb vector of Real

metal.mask.ellipses.yb vector of Real
x and y coordinates of the second focus of the ellipses. Each vector must contain *metal.mask.ellipses.number* of elements.

metal.mask.ellipses.sum vector of Real
The sum of distances between the point on the circumference of the ellipse and the two foci, i.e.,

$$S(x, y) = \sqrt{(x - x_a)^2 + (y - y_a)^2} + \sqrt{(x - x_b)^2 + (y - y_b)^2}$$

metal.mask.triangles.number int
Number of triangles that are going to be defined.

metal.mask.triangles.xa vector of Real

metal.mask.triangles.ya vector of Real
x and y coordinates of point A of each triangle. Each vector must contain *metal.mask.triangles.number* of elements.

metal.mask.triangles.xb vector of Real

metal.mask.triangles.yb vector of Real
x and y coordinates of point B of each triangle. Each vector must contain *metal.mask.triangles.number* of elements.

metal.mask.triangles.xc vector of Real

metal.mask.triangles.yc vector of Real
x and y coordinates of point C of each triangle. Each vector must contain *metal.mask.triangles.number* of elements.

Example

```
metal.mask.boxes.number        =  2
metal.mask.boxes.x_lo          = 30 60
metal.mask.boxes.y_lo          = 30 30
metal.mask.boxes.x_hi          = 40 70
metal.mask.boxes.y_hi          = 70 70
#
metal.mask.cylinders.number    =  3
metal.mask.cylinders.xc        = 50 60 70
```

```
metal.mask.cylinders.yc        = 50 60 70
metal.mask.cylinders.rc        = 10  5  5
#
metal.mask.triangles.number    =  2
metal.mask.triangles.xa        = 30 40
metal.mask.triangles.ya        = 30 30
metal.mask.triangles.xb        = 70 80
metal.mask.triangles.yb        = 30 30
metal.mask.triangles.xc        = 50 60
metal.mask.triangles.yc        = 70 70
```

Here we have 2 boxes, 3 circles and 2 triangles. The first box is defined by two points, $(30, 30)$ and $(40, 70)$. The second box is defined by points $(60, 30)$ and $(70, 70)$. The first of the three circles has its center at $(50, 50)$ and radius 10. The second circle has its center at $(60, 60)$ and radius 5 and the third circle has its center at $(70, 70)$ and radius 5. We also have two triangles that actually overlap. The first one is defined by three points $A = (30, 30)$, $B = (70, 30)$ and $C = (50, 70)$, and the second one is defined by $A = (40, 30)$, $B = (80, 30)$ and $C = (60, 70)$.

To kill the mask it is sufficient to just set the numbers of mask boxes, cylinders and triangles to zeros, leaving the coordinates unchanged.

The last subgroup of the *metal* group lets us specify where we *do* want a metal and what kind of metal. The semantics are very similar to the mask definition semantics with two differences: (1) we skip the word *mask* and (2) there is an additional keyword for each of the figures that specifies the medium number, where the medium number is as defined in the media definition section above.

The actual keywords are as follows.

metal.boxes.number int

metal.boxes.x_lo vector of Real

metal.boxes.y_lo vector of Real

metal.boxes.x_hi vector of Real

metal.boxes.y_hi vector of Real

metal.boxes.medium vector of int

metal.cylinders.number int

metal.cylinders.xc vector of Real

metal.cylinders.yc vector of Real

metal.cylinders.rc vector of Real

metal.cylinders.medium vector of int

metal.rings.number int

metal.rings.xc vector of Real

metal.rings.yc vector of Real

metal.rings.r_lo vector of Real

metal.rings.r_hi vector of Real

metal.rings.medium vector of int

metal.ellipses.number int

metal.ellipses.xa vector of Real

metal.ellipses.ya vector of Real

metal.ellipses.xb vector of Real

metal.ellipses.yb vector of Real

metal.ellipses.sum vector of Real

metal.ellipses.medium vector of int

metal.triangles.number int

metal.triangles.xa vector of Real

metal.triangles.ya vector of Real

metal.triangles.xb vector of Real

metal.triangles.yb vector of Real

metal.triangles.xc vector of Real

metal.triangles.yc vector of Real

metal.triangles.medium vector of int

Example

```
metal.cylinders.number         =  3
metal.cylinders.xc             = 20 40 70
metal.cylinders.yc             = 20 40 70
metal.cylinders.rc             =  5  5 10
metal.cylinders.medium         =  1  2  1
```

Here we have 3 circles defined. The first one is centered at $(20, 20)$, its radius is 5 and it is made of metal number 1. The second circle is centered at $(40, 40)$, its radius is 5 and it is made of metal number 2. Finally, the third circle is centered at $(70, 70)$, its radius is 10 and its made of metal number 1.

Different figures may overlap. If two figures representing different metals overlap, the last one wins in terms of metal assignment. If figures are of different shapes then cylinders win over boxes and triangles win over cylinders. A mask always wins over any metal shape. This is why it is called *a mask*.

The *tag* group

The next two groups, i.e., the *tag* group and the *refine* group are related to each other and relevant only if a construction of a multi-grid has been requested.

Program **shapes** can be run in a single level mode, i.e., without the multi-grid, and this is always preferable. The reason for this is that (1) computation on a multi-grid is costly and (2) multi-grid introduces noise and instabilities.

So why bother with a multi-grid at all? Well, sometimes it is useful. It is useful when we need very high resolution at certain locations only and when it would be prohibitively costly to impose the same resolution on the whole computational domain. Admittedly this is seldom the case in 2 dimensions, which is why multi-grid in this 2-dimensional version of **shapes** is a bit of an overkill. But this happens more often in 3 dimensions and this code is a prelude to a 3 dimensional code that is yet to follow.

A multi-grid can be generated dynamically - this is the Adaptive Mesh Refinement (AMR) technique, or statically. AMR is not really necessary for the type of problems that **shapes** is designed to simulate, but it is available. A static generation of a multi-grid is preferable and results in cheaper time-stepping, because regridding is a very expensive procedure. It is also possible to have a combination of static and dynamic multi-gridding in **shapes.**

Multi-gridding works by generating progressively finer meshes in selected locations. The meshes are organized into a vector with each mesh referred to as a *level*. Level 0 is the coarsest mesh that is defined in the *level0* group. Then we have level 1, which must be embedded entirely within level 0, level 2, which must be embedded entirely within level 1, and so on. The levels sit inside each other like Russian dolls. Each level may consist of disconnected patches.

The procedures that build a multi-grid work by looking at level 0 grid cells first and tagging them for refinement if need be. The cells get refined, generating a level 1 grid in the process. The procedures are then repeated within the level 1 grid, tagging its cells and refining them so as to generate level 2 grid, and so on until the whole hierarchy of grids is built. The hierarchy stops when the maximum desired number of levels is reached. This number is passed through the following keyword.

tag.max_number_of_levels int

Setting this number to 1 disables the multi-grid building procedure and multi-grid computations.

If multi-gridding has been requested by setting *tag.max_number_of_levels* to, say, 3 or 4, then the multi-grid generation procedures look at the next three keywords.

tag.on_location int

tag.on_diffs int

tag.on_values int

Each of these is a logical switch. When set to zero it disables tagging *on location* or *on diffs* or *on values*. When set to one it enables tagging. When all these switches are set to zero then no multi-grid is built, even if *tag.max_number_of_levels* is greater than 1.

Tagging *on location* means that cells are tagged for refinement depending on where they are. Tagging *on diffs* means that cells are tagged for refinement if energy density in the cells changes faster than a certain threshold value. Tagging *on values* means that cells are tagged for refinement if energy density in the cells exceeds a certain threshold value.

When tagging *on diffs* or *on values* is on, then the user must provide vectors of thresholds, i.e., a vector of values that will be used as thresholds within subsequent levels. The keywords here are

tag.diff_thresholds vector of Real

tag.value_thresholds vector of Real

Example

```
tag.max_number_of_levels = 4
tag.on_diffs             = 1
tag.diff_thresholds      = 0.02 0.05 0.08
tag.on_values            = 1
tag.value_thresholds     = 0.03 0.08 0.12
```

In this example we have requested 4 levels, level 0, level 1, level 2 and level 3. We have also requested that tagging cells for refinement be done by looking at how fast energy density changes within the cells and how large the value of energy density actually is. And so, cells of level 0 will be tagged if energy density within the cell changes by 0.02 within Δt_0. Cells of level 1 will be tagged if energy density within the cell changes by 0.05 within Δt_0 (yes, this is because energy density is computed for all levels every level 0 time step, Δt_0, only). And, finally, cells of level 2 will be tagged if energy density within the cell changes by 0.08 within Δt_0. Cells within level 3 will not be tagged, because we don't want level 4. This is why there are only 3 Reals in the vector, not 4. Additionally cells of level 0 will be tagged for refinement if the value of energy density within the cell exceeds 0.03, cells of level 1 will be tagged if the value of energy density within the cell exceeds 0.08 and cells of level 2 will be tagged for refinement if energy density within the cell exceeds 0.12.

If tagging is also activated *on location* then the tagging procedures look at the following list of keywords.

tag.mask.boxes.number int

tag.mask.boxes.x_lo vector of Real

tag.mask.boxes.y_lo vector of Real

tag.mask.boxes.x_hi vector of Real

tag.mask.boxes.y_hi vector of Real

tag.mask.cylinders.number int

tag.mask.cylinders.xc vector of Real

tag.mask.cylinders.yc vector of Real

tag.mask.cylinders.rc vector of Real

tag.mask.triangles.number int

tag.mask.triangles.xa vector of Real

tag.mask.triangles.ya vector of Real

tag.mask.triangles.xb vector of Real

tag.mask.triangles.yb vector of Real

tag.mask.triangles.xc vector of Real

tag.mask.triangles.yc vector of Real

tag.boxes.number int

tag.boxes.x_lo vector of Real

tag.boxes.y_lo vector of Real

tag.boxes.x_hi vector of Real

tag.boxes.y_hi vector of Real

tag.cylinders.number int

tag.cylinders.xc vector of Real

tag.cylinders.yc vector of Real

tag.cylinders.rc vector of Real

tag.triangles.number int

tag.triangles.xa vector of Real

tag.triangles.ya vector of Real

tag.triangles.xb vector of Real

tag.triangles.yb vector of Real

tag.triangles.xc vector of Real

tag.triangles.yc vector of Real

The semantics of these constructs are identical to the semantics of similar constructs discussed in the section about the *metal* group. But here, instead of applying metals, we apply subgridding. As was the case with metals, applying a *tag* mask *disables* the generation of a sub-grid in this area. The mask always wins over other shapes.

Observe that if *tag.boxes.number*, *tag.cylinders.number* and *tag.triangles.number* are all zero then no *on location* tagging will be performed, even if *tag.on_location* is one.

The *refine* **group**

Once the cells of a given level have been tagged they need to be refined. **shapes** uses a fixed refinement ratio of 2, i.e., the distance between adjacent nodes of level n is equal to half the distance between adjacent nodes of level $n-1$. The time refinement between adjacent levels is 3, i.e., the time step used in level n is equal to one third of the time step used in level $n-1$, unless the user has requested the same time step for all levels explicitly by switching off the *iterate.use_substep* option.

There are four parameters in the *refine* group that can be used to provide additional specifications as to how the refinement should be done.

refine.fill_ratio Real

This parameter tells Chombo how tightly to wrap subgrids around the shapes provided by the user. The generated subgrids will be tightest when this parameter is set to 1.0. But this may produce a large number of very small boxes of grid points on top of some larger boxes. The boxes of grid points we are talking about here are the atomic unit of Chombo parallelization. Each CPU gets either nothing or a box of grid points to work on or several boxes. If some boxes are very small and other very large, some CPUs may get very little work whereas other CPUs may get overloaded. The fit will be most relaxed, i.e., the grids will be oversized and boxes will be large, when this parameter is set to 0.0. Setting it to, e.g., 0.5, results in a pretty oversized and relaxed grid already. I seldom use anything lower than 0.8.

refine.block_factor int

This is the smallest size of a box, in the generated grid cells, that Chombo will use when building a refined grid. From my experience - this number should be a power of 2. If set to, e.g., 3, 5 or 10 it may result in a subgrid build failure. When set, e.g., to 4, the smallest box that Chombo is going to generate will be 4x4.

refine.buffer_size int

This parameter tells Chombo how deep to nest a finer level within a coarser level. The distance between, say, level 3 border and level 2 border will be *refine.buffer_size* level 2 cells.

refine.max_size int

The maximum size of a box that is going to be generated. When set, e.g., to 100, the largest boxes that Chombo is going to generate will be 100×100.

Example

```
refine.fill_ratio   =  1.0
refine.block_factor =  2
refine.buffer_size  =  8
refine.max_size     = 50
```

The meaning of this input is as follows. Generate subgrids as tight as you can around the shapes provided. Let the smaller boxes in the generated sub-grids be 2×2 and the largest ones 50×50. Generate an 8-cell wide margin between the borders of adjacent levels.

The *spectral* **group**

shapes will calculate spectral response for E_x, E_y, H_z and energy density, $(H^2 + E^2)/2$, on request, for any number of frequencies ω. This is done by running the following summations and evaluating the amplitude and the phase angle of the response for a selected field $f(x, y, t)$:

$$\hat{f}_r(x, y, \omega) = \sum_{k=1}^{k=n} f(x, y, t_k) \cos(\omega t_k) \Delta t$$

$$\hat{f}_i(x, y, \omega) = \sum_{k=1}^{k=n} f(x, y, t_k) \sin(\omega t_k) \Delta t$$

$$\left| \hat{f}(x, y, \omega) \right| = \sqrt{\hat{f}_r^2(x, y, \omega) + \hat{f}_i^2(x, y, \omega)}$$

$$\Phi(x, y, \omega) = \arctan\left(\hat{f}_i(x, y, \omega), \hat{f}_r(x, y, \omega) \right)$$

where k is the step number and n is the number of time steps, Δt, performed so far.

The computation is enabled by setting the

spectral.response int

switch to 1. It is disabled by setting it to 0.

If *spectral.response* is activated then the user must also specify the number of frequencies and the frequencies themselves.

spectral.number_of_frequencies int
A number of angular frequencies for which the spectral response is to be evaluated.

spectral.frequencies vector of Real
A vector of *angular* frequencies in natural units for which the spectral response is to be evaluated.

Example

```
spectral.response              = 1
spectral.number_of_frequencies = 4
spectral.frequencies           = 0.1570 0.2093 0.3140 0.6280
```

The frequencies must be provided in the natural units like everything else in the program. Also, note that they are *angular* frequencies, i.e., $2\pi/T$, where T is the period of vibration. In the natural units T of a harmonic wave is the same as its length λ, so the angular frequency that matches that of a harmonic wave of length λ is $2\pi/\lambda$. In this example, the frequencies are $2\pi/40$, $2\pi/30$, $2\pi/20$ and $2\pi/10$.

The actual fields for which spectral response is going to be computed are specified in the *output* group discussed in the next section. If no fields are specified for spectral response then obviously no spectral response computations will take place, even if *spectral.response* is set to 1 and the vector of angular frequencies specified.

The *output* group

shapes can generate output in HDF5 or Gnuplot formats. The Gnuplot format can be used only in the sequential version of the program, otherwise Gnuplot format directives are ignored. The HDF5 format can be used always. **shapes** can be asked to output data in *both* formats too.

The keywords used for this are

output.gnuplot int
Set it to 1 to enable Gnuplot output. Set it to 0 to disable Gnuplot output.

output.hdf5 int
Set it to 1 to enable HDF5 output. Set it to 0 to disable HDF5 output.

Disabling both HDF5 and Gnuplot outputs disables all data generation altogether. The program will still run though and it will print various diagnostics on standard output or on the MPI output files if run in parallel and if such diagnostics have been requested. This mode of operation can be used for debugging.

In the Gnuplot output mode the program generates separate files for each field. The Gnuplot data file names are as follows

```
<field_name>_<level_number>_<snapshot_number>.dat
```

where *field_name* can be one of

```
Dx Dy Ex Ey E Hz Energy Distrib_Dx Distrib_Dy
```

Additionally for the spectral analysis output we have

```
Ex_amplitude_<frequency>       Ex_phase_<frequency>
Ey_amplitude_<frequency>       Ey_phase_<frequency>
E_amplitude_<frequency>        E_phase_<frequency>
Hz_amplitude_<frequency>       Hz_phase_<frequency>
Energy_amplitude_<frequency>   Energy_phase_<frequency>
```

For example *Ex_3_074.dat* will be a file that contains E_x data for level 3 and snapshot 74, whereas *Ex_amplitude_0.1570_3_074.dat* will be a file that contains $\left|\hat{E}_x\right|$ data at $\omega = 0.1570$ for level 3 and snapshot 74.

The data in the Gnuplot files can be displayed by using the Gnuplot *splot* command. For example

```
set xrange[10:90]
set yrange[10:90]
```

```
splot "Ex_0_074.dat" with lines
```

Gnuplot data files contain extensive annotations in the headers, in which the full computational system is described. For example

```
# program: shapes, function: write_gnuplot_data
# header:
#    program author: Zdzislaw (Gustav) Meglicki, Indiana University
#    @Id: shapes.cpp,v 1.21 2005/12/22 18:41:29 gustav Exp @
#    @Id: shapes.h,v 1.64 2005/12/17 23:41:59 gustav Exp @
#    @Id: levels.cpp,v 1.88 2005/12/22 22:14:34 gustav Exp @
#    @Id: io.cpp,v 1.18 2005/12/22 17:37:10 gustav Exp @
#    @Id: update.f,v 1.42 2005/12/18 18:41:10 gustav Exp @
#    system kernel:  CYGWIN_NT-5.1.1.5.18(0.132/4/2).2005-07-02 20:30
#    machine:        i686
#    node:           woodlands
#    time of dump:   Thu Dec 22 18:25:35 2005
# Signal injection group:
#    x_lo:           20.000000
#    y_lo:           20.000000
#    x_hi:           80.000000
#    y_hi:           80.000000
#    mode:           7 (Gaussian envelope quadratic chirp)
#    t0:             300.000000
#    lambda:         10.000000
#    sigma:          60.000000
#    alpha:          0.000000
#    beta:           0.001200
#    vx:             -1.000000
#    vy:             0.000000
# Media group:
#    No media defined
# Data group:
#    data for:       Hz
#    level:          0
#    label:          173
#    time_e:         346.000000
#    time_h:         346.062500
#    delta_t:        0.125000
#    delta_t_0:      0.125000
#    xmin:           0.000000
#    xmin_0:         0.000000
#    xmax:           99.500000
#    xmax_0:         99.500000
#    ymin:           0.000000
#    ymin_0:         0.000000
#    ymax:           99.500000
#    ymax_0:         99.500000
#    delta_x:        0.500000
#    delta_x_0:      0.500000
#    delta_y:        0.500000
#    delta_y_0:      0.500000
#    data minimum:   -1.010560
#    global minimum: -1.010560
#    data maximum:   1.011332
```

```
#    global maximum: 1.011332
# data:
# x:       y:        Hz:
  -0.500   -0.500    0.00000
   0.000   -0.500    0.00000
   0.500   -0.500    0.00000
   ...
```

Gnuplot data files can be converted to GIF animations as follows.

1. For each data file from the series to be animated, e.g., *Hz_0_005.dat* set Gnuplot terminal to *png*, set Gnuplot output to *Hz_0_005.png*, define a title for the plot and then *splot* the data file. Gnuplot will write an image in the *png* format on *Hz_0_005.png*.
2. Convert each *png* image first to the *gd2* image with *pngtogd2* and then convert it to a *gif* file with *gd2togif*.
3. Finally assemble all the *gif* files into a movie by calling *gifsicle*, for example

```
gifsicle --delay 20 --loopcount Hz_0_[0-9][0-9][0-9].gif > Hz_0_movie.gif
```

There are three scripts provided with the source in the *script* subdirectory, *dattomovie.sh*, *write_gif_files.sh* and *pngtomovie.sh*, that show how to automate this process.

HDF5 data files collate all fields selected for a given snapshot at all levels on a single file called *fields_<snapshot_number>.hdf5*. The field data is written first in a special machine independent IEEE approved format, not as text, and it is additionally *compressed*, so the files take much, much less space than Gnuplot files, which are all plain, human readable text files and are therefore huge. On the other hand extracting data from the HDF5 files and doing anything with it is a major pain. A special tool called *ChomboVis* must be used to post-process and visualize data from the HDF5 files dumped by **shapes.** HDF5 files contain some additional information in the headers, but not as much as Gnuplot files. This may change in future versions of the program. There should be no side-effects to this change.

Now we get to specify the actual fields that are dumped. This is done by the following keywords.

output.Dx int
: Output the D_x field. 1 enables, 0 disables. The D_x field is dumped for the centers of the cells, not for the sides, where it resides during the computations. This means that it is space interpolated between $D_x(x, y - \Delta y/2)$ and $D_x(x, y + \Delta y/2)$ before dumping.

output.Dy int
: Output the D_y field. 1 enables, 0 disables. The D_y field is dumped for the centers of the cells, not for the sides, where it resides during the computations. This means that it is space interpolated between $D_y(x - \Delta x/2, y)$ and $D_x(x + \Delta x/2, y)$ before dumping.

output.Ex int
: Output the E_x field. 1 enables, 0 disables. This field is space interpolated before dumping the same way as D_x.

output.Ey int
: Output the E_y field. 1 enables, 0 disables. This field is space interpolated before dumping the same way as D_y.

output.E int
: Ouput the $E = \sqrt{E_x^2 + E_y^2}$ field. 1 enables, 0 disables.

output.Hz int
: Output the H_z field. 1 enables, 0 disables. The H_z field is output *for the same time slice* as the $\vec{E}$ field. The data is actually interpolated between $H_z(t - \Delta t/2)$ and $H_z(t + \Delta t/2)$ prior to the dump.

output.Energy int
: Ouput the energy density. 1 enables, 0 disables. The energy density is output *for the same time slice* as the $\vec{E}$ field. Time interpolated H_z data (see above) is used in the computation. It is also

output for the centers of the cells, so that space interpolated $\vec{E}$ data is used in the computation.

output.Distrib_Dx int
Output the distribution of metal on the D_x grid sites. 1 enables, 0 disables. Because the distribution of metal does not change with time, it is pointless to dump it for every snapshot. This option is meant to be used for initial runs only. Once the user is happy with the metal distribution and can get the picture, this option should be disabled.

output.Distrib_Dy int
Output the distribution of metal on the D_y grid sites. 1 enables, 0 disables. Same comments apply.

output.Ex_ft int
Output the $\hat{E}_x$ field for frequencies specified by the *spectral.frequencies* vector. 1 enables, 0 disables. Cell centered E_x data is used in this computation.

output.Ey_ft int
Output the $\hat{E}_y$ field for frequencies specified by the *spectral.frequencies* vector. 1 enables, 0 disables. Cell centered E_y data is used in this computation.

output.E_ft int
Output the $\hat{E}$ field for frequencies specified by the *spectral.frequencies* vector. 1 enables, 0 disables.

output.Hz_ft int
Output the $\hat{H}_z$ field for frequencies specified by the *spectral.frequencies* vector. 1 enables, 0 disables. Time centered H_z data is used in this computation.

output.Energy_ft int
Output the energy density spectral response field for frequencies specified by the *spectral.frequencies* vector. 1 enables, 0 disables. Cell and time centered data is used in this computation.

output.tags int
Output cells tagged for refinment. This option works with Gnuplot output only. 1 enables, 0 disables.

output.boxes int
Output subgrids. This option works with Gnuplot output only. Subgrids are *always* included in the HDF5 output. 1 enables, 0 disables.

output.some_levels_only int
Sometimes we are interested in the highest level only, or in a range of levels. This option enables (1) or disables (0) such output. If enabled, it must be followed by **output.from_level** and **output.to_level** keywords.

output.from_level int
When **output.some_levels_only** is activated dump levels beginning with this one (inclusive).

output.to_level int
When **output.some_levels_only** is activated dump levels up to this one (inclusive).

Example

```
output.gnuplot                = 1
output.hdf5                   = 0
output.Dx                     = 0
output.Distrib_Dx             = 0
output.Dy                     = 0
output.Distrib_Dy             = 0
output.Ex                     = 1
output.Ex_ft                  = 1
output.Ey                     = 1
output.Ey_ft                  = 1
```

```
output.Hz                       = 0
output.Hz_ft                    = 0
output.Energy                   = 0
output.Energy_ft                = 0
output.tags                     = 0
output.boxes                    = 0
output.some_levels_only         = 1
output.from_level               = 3
output.to_level                 = 4
```

Here we have requested Gnuplot style output, but HDF5 output is disabled. We are going to dump data for $\vec{E}$ and for its spectral response, but not for $\vec{H}$ or energy. Only data for levels 3 and 4 will be dumped. Note that if the program is run in parallel, no data will be dumped at all, because only HDF5 data can be dumped for parallel runs.

EXAMPLES

Example input files can be found in

/soft/apps/packages/photonic-packages/Confs

NOTES

At this stage the 2-dimensional version of **shapes** is pretty much complete. I hope to maintain the input file comptability from this point onwards as small modifications and improvements are made to the program.

BUGS

Some design shortcomings that *may* be rectified in due course are listed here.

There is no way to specify if a given figure should be used with or without its boundary.

There is no way to specify multiple concentric rings, although a single ring can be specified with a circle and a mask.

Material (and geometric) parameters cannot be input in eV or SI units.

The keyword **cylinders** is used instead of **circles** and **boxes** is used instead of **rectangles.**

Very complex layouts may be difficult to input. An interactive graphical user interface could be provided that would generate a **shapes** input file automatically.

The HDF5 file headers do not contain as much information as the Gnuplot file headers.

AUTHOR

Zdzislaw (Gustav) Meglicki and his cats, Bambosz and Sofa.

SEE ALSO

shapes(1)

Shapes User Guide