

?88,.d88b, d8888b d8888b 88bd8b,d88b .d888b,
'?88' ?88d8P' ?88d8b_,dP 88P''?8P'?8b ?8b,
88b d8P88b d8888b d88 d88 88P '?8b
888888P''?8888P''?888P'd88' d88' 88b'?888P'
88P'
d88
?8P

A Programmable Optimizing Electro-Magnetic Simulator

Release 1.97

November 21, 2010

Phil Hobbs

ElectroOptical Innovations
Briarcliff Manor, NY

POEMS:

A Programmable Optimizing
Electro-Magnetic Simulator

Release 1.97

November 21, 2010

Phil Hobbs

ElectroOptical Innovations
Briarcliff Manor, NY

Chapter 1.	
Introduction	1
1.1. Motivation (1)	
2. HOW POEMS WORKS	6
2.1. Program Organization (6); 2.2. The Front-End Script: poems.cmd (6); 2.3. The FDTD Engine: FIDO/TEMPEST (7); 2.4. The Postprocessor: EMPOST (7); 2.5. The Visualization System: VIS5D (8); 2.6. Cluster Control (8)	
3. USING POEMS	10
3.1. Command Reference (10); 3.2. The Computational Domain (31); 3.3. OBJECTS (32); 3.4. MATERIALS (33); 3.5. SOURCES (33); 3.6. Optimization (37); 3.7. Predefined Constants (41); 3.8. Predefined Mathematical Functions (43); 3.9. Analytical Pupil Functions (48); 3.10. Material Parameter Functions (48); A. TEMPEST and General FDTD Information (49)	
Appendix A.: V-Antenna Optimization Run	51
A.1. POEMS INPUT: DIPOLE2I.PAR (52); A.2. tempest Input File: DIPOLE2I.PAR.IN (62); A.3. Postprocessor orders: DIPOLE2I.ORDERES (65); A.4. Run Results: DIPOLE2I.SIMPLEX (75)	
Appendix B.: Configuration	81
B.1. FDTD and TEMPEST (81); B.2. REXX (83); B.3. X Window System Configuration (83); B.4. Release Notes (84)	

Chapter 1. Introduction

1.1. Motivation

POEMS is a tool for analyzing and synthesizing wavelength-scale electromagnetic devices. It is similar to existing 3-D vector electromagnetic simulators in many ways; it takes an input file and produces **E** and **H** field values, pictures, and scalar outputs such as energy flux through surfaces. POEMS can simulate such structures as waveguide tapers and bends, photonic crystals, antennas, resonators, and couplers. It is competitive in power and features for such uses.

However, its capabilities go well beyond there. The name POEMS is short for *Programmable Optimizing Electromagnetic Simulator*. What makes POEMS special are three attributes:

1. Its technological orientation. You can calculate the total power dissipated in a resonator, compute the efficiencies of Gaussian and Airy beams in driving a waveguide horn, plot the far-field pattern of a scatterer, or give a grating a half-wave of spherical aberration, in one or two source lines each. The simulation output can be displayed and explored using Vis5D, an advanced interactive visualization program that comes with the POEMS distribution. The simulation geometry can be exported as a CATS .ctxt or Autocad .DXF file for import into mask design or mechanical CAD packages.
2. Its readability. The previous point makes POEMS sound like APL or Perl, but it isn't at all. People can easily read and understand one another's POEMS files (or their own from six months ago). Function parameters can be symbols or mathematical expressions, and are passed in the form of assignments, e.g. `phase = pi/6` or `width = sqrt(area)` rather than as long lists of floating-point numbers that all have to be in the right order. Parametrized macros can be used to reduce the amount of repeated code for similar operations. The effect is to make POEMS more like a math program and less like a simulator.
3. Its power and generality. It is one thing to analyze the performance of a design once it is finished, and quite another to synthesize a good design for a particular purpose. POEMS is an *optimizing* simulator. Given suitable starting points, it will automatically adjust any parameters you specify to optimize any criterion you give it. The parameters and merit functions are completely user-specified—many properties of the device may depend on each parameter, if desired. For example, you can optimize the shape of an antenna to maximize the power dissipated in the load resistance, for plane wave illumination, or optimize the aberration coefficients of a grating to improve a free-space to waveguide transition. There are lots of programs to analyze given structures.

Of course, there is one very good reason why such a capability has not been available before: it can be quite slow. One simulation can take minutes or hours to run, so an optimization requiring many runs may take quite a while. While this is still a cogent objection in many cases, POEMS' ability to scale to large clusters can make this pretty snappy. Even without a cluster, the continued improvements in personal computer CPU speed and memory size allow nontrivial multi-parameter optimizations to be run on a laptop in a few hours, with little or no supervision. Given the economic importance of many of these devices, e.g. phase shift masks and optical waveguide devices, there is now a large class of problems for which an optimizing FDTD simulator is a useful tool. This is particularly true when the simulator can run seamlessly on one machine or a large cluster of machines of different types and architectures, as POEMS can.

Furthermore, the same techniques designers use to guide existing simulators, e.g. physical and analytical models, can be used with POEMS, with an order-of-magnitude decrease in the amount of time spent baby-sitting the simulator.

The current release of POEMS, V 1.63, does almost all of these things already, and more are under development.

1.1.1. Philosophy

The idea of POEMS is to keep the design problem in view, and to make the program fade into the background. This doesn't need fancy user interfaces so much as freedom from limitations and constant manual-reading. This philosophy drove the design, leading to these goals:

Clarity:

- Accept human-readable input with no unnecessary parameter order dependencies;
- Use mnemonic names;
- Provide accurate and specific error messages
- Understand optical terms, e.g. amplitude and phase, aberration coefficients, power dissipation, efficiency, mode matching with commonly used pupil functions e.g. Gaussian and Airy (uniform pupil);
- Provide high-level geometric constructs, e.g. gratings and smooth curved tapers and bends;
- Specify dimensions the way you'd measure with calipers—round correctly and avoid worries about counting from 0 or from 1;
- Automate fiddly things that go wrong easily, e.g. configuring the perfectly-matched layer (PML) absorption directions or figuring out n and k for a normal conductor

Power:

- Use the best existing open-source software, e.g. FFTW and VIS5D
- Work on many platforms, at least Linux, OS/2, and Windows (The author is an OS/2 diehard but recognizes the quixotic character of this)
- Provide advanced visualization tools: bitmaps, animations, and (especially) 3-

dimensional visualization via Vis5D

and, most importantly,

-Provide a powerful way to optimize structures for a given purpose, e.g. couplers, masks, antennas, and so forth.

To avoid wasted effort, as much as possible of the information generated by the simulation needs to be kept for further use and for sanity checks along the way. It should also be possible to stop an optimization run in the middle and restart it without losing the previous results, using the `RESTART` command line option (see Section 3.1.1). The current release of POEMS does all of these things.

1.1.2. Structure

POEMS consists of a front-end script that handles all the housekeeping, optimization, and interface duties, a simulation engine, a postprocessor to turn raw values of **E** and **H** into useful output, and a visualization system based on Vis5D. The simulation engines are currently TEMPEST, a finite-difference time-domain (FDTD) simulation code developed by Alfred Wong and Tom Pistor at the University of California at Berkeley, whose single-processor form is widely available in C source; and FIDO, a plug-compatible program written from scratch. FIDO accepts the subset of the TEMPEST input file format used by POEMS, with some extensions, so that single-processor simulation can be run on both, and the results compared. FIDO is a significantly more advanced design than TEMPEST, since it precomputes a strategy for the computation instead of just putting a huge switch statement inside a triply-nested `for()` loop. In addition, it is multithreaded, which allows efficient utilization of symmetric multiprocessor (SMP) machines without additional user effort. On a dual-processor 2.8 GHz Xeon machine running Windows XP, FIDO is about 2.5x faster than TEMPEST on a typical problem--2x for two CPUs, and 25% faster on a per-CPU basis. With this release, FIDO can now run on clusters of Linux, Windows, or OS/2 clusters connected over a TCP/IP network. There is some performance penalty for this, mainly due to communications latency, but for simulations large enough that each time step takes at least 0.5 second, the scaling is excellent.

The user interacts with the front-end script via a high-level input file. Commands in the input file are parsed in the order written. For example, you can't use a variable at a point above its definition in the file.

1.1.3. Optimization

The POEMS optimizer is currently a vanilla Nelder-Mead downhill simplex algorithm, similar to the Numerical Recipes `AMOEB` routine. (A simplex is an N-dimensional figure with N+1 vertices--examples are triangles and tetrahedrons.) Nelder-Mead stays out of trouble pretty well in optimizing continuous functions, unless the simplex gets pathologically long and skinny, in which case it complains. Most of the time, though, we're optimizing simulation geometry, i.e. which block gets which material. Object dimensions come only in integer multiples of the local cell size, so rounding occurs before the object is generated. Changing the requested size has no effect whatever until it crosses the centre line of a cell, causing a discontinuous change in the actual object size,

and hence in the simulation results. On small scales, therefore, the partial derivatives of any penalty function will be zero in most places, with delta-functions sprinkled round, and the penalty function surfaces exhibit multidimensional cliffs and canyons. This is why we're not using one of those fancy variable metric optimizers. POEMS has no way of exploring the bottom of a canyon it never encounters, so for simulations with significant economic importance, it's worth restarting the simulator a few times using the `RESTARTS` parameter of the `COMPUTE` statement. (For more discussion, see e.g. W. H. Press *et al*, *Numerical Recipes in C*, 2nd Ed., Cambridge (UK), Cambridge University Press, 1991.

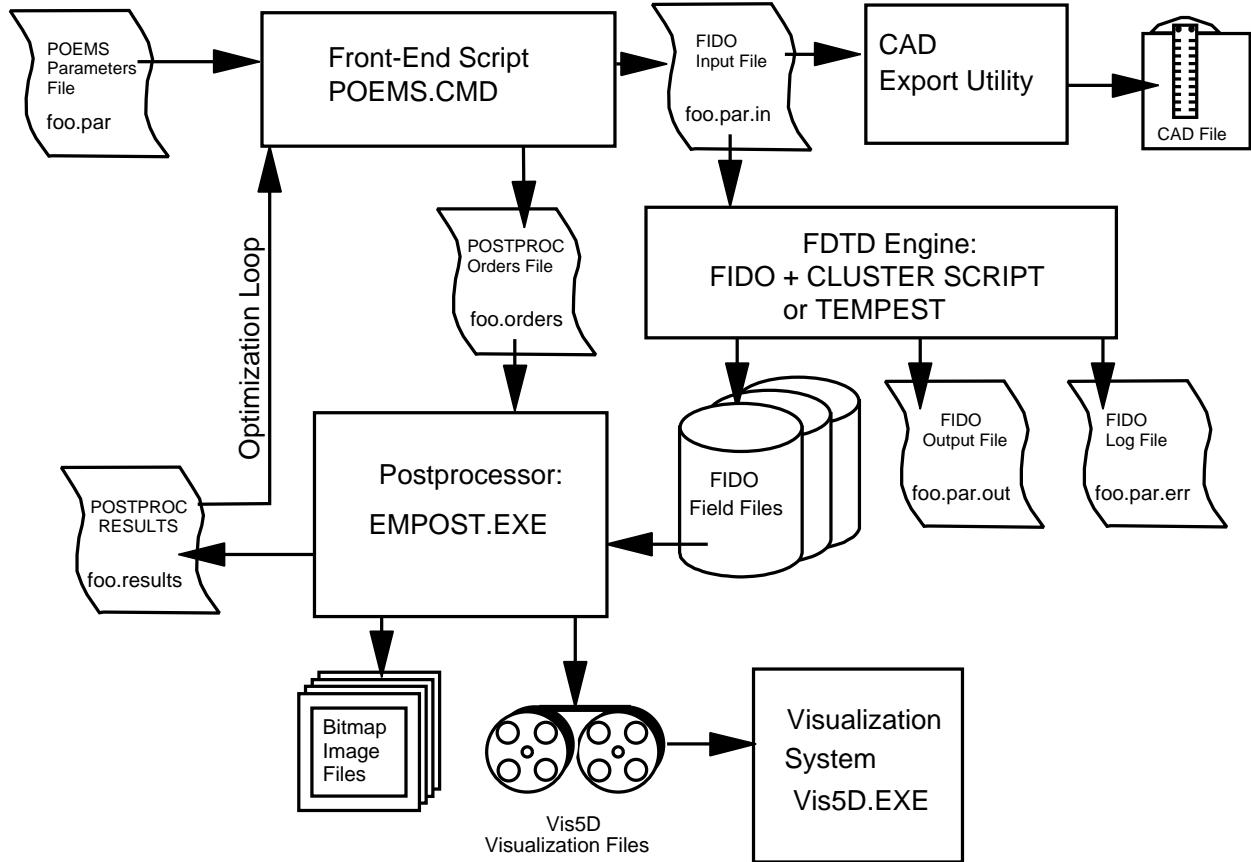


Figure 2.1: POEMS system organization

2. HOW POEMS WORKS

2.1. Program Organization

The POEMS system consists of six parts: the front-end script (`poems.cmd`), which controls the whole process, communicating with the user, doing the less CPU-intensive tasks (such as CAD file generation) and controlling the rest; the FDTD simulation engine (FIDO or TEMPEST), which takes a simulation description file and produces binary files of simulated **E** and **H** field values; the postprocessor (EMPOST), which takes those huge binary files and produces secondary binary data and pictures, including VIS5D interactive visualizations; Vis5D itself, an open-source data visualization program running under the X Window system; and, optionally, a cluster control script (`fidossh`) that distributes parameters, starts the simulator instances, and collects data afterwards. We'll examine these in turn.

2.2. The Front-End Script: `poems.cmd`

The front end script `poems.cmd` is the part of POEMS that the user will interact with (almost exclusively). It parses the input file, performs error checking, and handles all the symbols and equations. It generates the intermediate files that are used as input by FIDO/TEMPEST and EMPOST, and produces log files and console output to keep the user informed as to the progress of the run. The script is written in REXX, whose advanced parsing capabilities and very flexible stem processing made it a natural choice. Speed is not a serious issue, since only a tiny part of the run time is spent in the REXX code; almost all is used in the FDTD simulation.

2.2.1. Script Operation

The concept underlying the script's design is that the user should be able to optimize or step anything he likes, in any combination. Therefore, user-defined functions, expressions, and variables are accepted anywhere an argument is required; as in a math program, these variables and functions can depend on each other in any fashion consistent with top-to-bottom parsing of the input file. This is straightforward in a single simulation run, but is a little more involved where stepping and optimization is being used: the program cannot make any assumptions about which simulation parameters can depend on the controlling variables. Accordingly, an optimization or stepping run is organized as follows.

- a. Enter SETUP mode
- b. Parse the input file and set all the variables. Note which variables are to be stepped or optimized (the *controlling* variables).
- c. Enter OPTIMIZE or STEP mode. For each iteration,
 - (i) Update the controlling variables for the current iteration
 - (ii) Parse the input file, setting all the variables except the controlling ones. This preserves all the dependencies.
 - (iii) Generate the intermediate files for FIDO/TEMPEST and EMPOST, based on the current values of all the variables.
 - (iv) Run FIDO/TEMPEST, capturing its console output to a file.
 - (v) If FIDO/TEMPEST fails, stop. If it succeeds, call EMPOST to generate the inputs to the

merit function, plus any binary file, list, or slice bitmaps requested. Doing this on each iteration takes little time and helps in supervising the run's progress.

(vi)Based on the computed merit function, update the optimization simplex. If convergence has occurred, exit. Otherwise, compute new values of the controlling variables for the next iteration and keep iterating. If the current point is the best so far, save all the bitmap, list, and mode files under another name.

Of course, in order for this to do anything useful, you have to specify a merit function (or penalty function for pessimists) that depends on the simulation output.

2.3. The FDTD Engine: FIDO/TEMPEST

POEMS started out life using TEMPEST as its main component. TEMPEST is a more or less vanilla single-processor FDTD engine, written originally for simulation of phase-shift masks, but quite widely used for a variety of applications. It is described in its own documents, which accompany the POEMS distribution. Due to the limitations of TEMPEST, in particular its lack of subgridding and multiprocessor capabilities, POEMS now relies principally on a specially written FDTD engine called FIDO, for FInite difference time DOrain. In broad outline, each of these programs parses an input file containing human-readable, hard-coded specifications of the simulation domain, boundaries, materials, objects, sources, and binary output files; constructs and runs a FDTD simulation as specified, stopping when the specified degree of convergence has been attained or the maximum cycles exceeded, and producing large binary files full of **E** and **H** values. TEMPEST comes with Matlab scripts to plot these simulated fields and do simple manipulations on them. Section A.2 has an example of a FIDO/TEMPEST input file generated by `poems.cmd`.

TEMPEST also has some more advanced capabilities, e.g. Fourier boundary conditions, far-field computation (via the `orders` output command), and more complicated source shapes, that POEMS ignores in favour of its own more general versions. This was done for reasons of usability and to avoid being tied too tightly to one particular simulator engine. One additional (and most important) attribute of TEMPEST is that it is well validated. Besides having been tested on problems whose analytical solution is known, it has done a good job for lots of people over several years. For this reason, FIDO was written to be a plug-compatible superset of TEMPEST: it takes the same input files, and can do the same simulations, but FIDO is about 50% faster on a per-processor basis and can do subgridded and clusterized simulations as well. Thus in new situations we can test our simulations using TEMPEST and then refine them using FIDO.

2.4. The Postprocessor: EMPOST

Large binary files full of **E** and **H** values are not very useful by themselves. The tool set provided by POEMS requires a lot of CPU-intensive calculations and binary manipulations that are much better done in C++ than in REXX. Accordingly, POEMS uses a postprocessor written in C++ to do most of its numerical work. EMPOST takes a human-readable but quite rigidly formatted `orders` file that specifies what is to be done on what data. An example is shown in Section A.3.

It is occasionally useful to run the postprocessor manually, e.g. when we want to change

from a linear to a logarithmic scale, change a colour palette, or something like that; simple changes like that are easily made directly to the generated file. EMPOST's calling syntax is

```
empost orders_file results_file
```

The second argument is the name of a file that EMPOST produces, containing numerical results of the named orders, e.g. integrals, fluxes, mode matching coefficients, and so on. These are in the form of assignment statements, and are parsed by POEMS when postprocessing is completed. POEMS then uses these to compute the value of the penalty function for the current iteration.

2.5. The Visualization System: VIS5D

VIS5D is an advanced visualization program originally written for meteorological data. It runs under the X Window System, which is native to Linux and other Unix derivatives, but which has to be added to Windows and OS/2. Windows users can install Hummingbird Exceed, which works well with VIS5D once all the arcane X parameters are set up. See Appendix B for a working sample X configuration.

The Vis5D conversion code in empost is based on a stand-alone program by Theodore G. van Kessel. In this release, it is fully integrated into EMPOST. Both animated and static Vis5D files are generated using the MOVIE3D statement.

2.6. Cluster Control

There are lots of ways to structure a cluster, lots of communications styles (such as the Message Passing Interface (MPI), and lots of cluster management systems such as the Sun Grid Engine (SGE). POEMS is not tied to any of these, but is easily adapted to them. The main script runs on a frontend machine. Inter-host communication requires no specific support other than a high capacity, low latency TCP/IP network. FIDO uses TCP/IP socket communication between fido subdomains running on different hosts, and local communication between subdomains running on the same host. To allow the user control, cluster control is not hardcoded into POEMS, but relies on an external script. The supplied script is `fidossh`, which uses a shared file system (e.g. NFS, XFS, or PVFS2) for communication and `ssh` for cluster node control. This design is suitable for clusters of up to perhaps 20 nodes, depending on filesystem performance. The high bandwidth host-to-host communication is organized in a distributed fashion between cluster hosts, so the frontend machine does not become a bottleneck in small and medium sized clusters. For larger clusters, FIDO can use a hierarchical supervision scheme, where a single frontend node is not forced to supervise hundreds or thousands of hosts, but the cluster script would have to be tailored for the application.

The downside of this flexibility is that the user has to apportion the work manually. Future versions of POEMS will help automate this, based on the CPU speed, number of cores, and amount of memory possessed by each host. Probably it will remain semiautomatic unless the subdomains can be made very small.

2.6.1. Parallel Processing

[Under construction]

FIDO is a powerful and versatile simulation engine, which can compute simulations using inhomogeneous cubic grids on uniprocessors, symmetric multiprocessors (SMPs), and clusters tied together with TCP/IP. From the POEMS user's point of view, SMPs act just like uniprocessors, except that an N-way SMP needs its work divided up into at least N chunks, and the load balancing is manual--usually it's easy. This section discusses multiprocessor operations; technical details are in Appendix C.

3. USING POEMS

Everything in POEMS is case-insensitive. Case is preserved but not significant in file names; specifying two names equal except for case will cause the first to be overwritten in Windows and OS/2 but not in Linux.

3.1. Command Reference

3.1.1. poems Command-Line Options

The calling syntax for POEMS is

```
poems parmfile <- option1 <option2 <option3>>>
where the options are one of DEBUG VERBOSE and RESTART.
DEBUG prints lots of debugging information,
VERBOSE prints more detail in the ordinary console output, and
RESTART causes POEMS to parse the specified simplex file and
restart the optimization run following the last completed iteration."
```

3.1.2. GLOBAL Group

ASSERT

Syntax: ASSERT *<expression>*

Allows the user to add parameter error checking to the simulations. ASSERT functions very much like the `assert()` macro in C. Each time the input file is parsed (i.e. at the beginning of the run and before each iteration of the optimizer or stepper), *<expression>* is evaluated. If the result is zero, the run is stopped and a specific error message printed.

Example: ASSERT SourceZ ≤ Zsize-Tpml-lambda

COMMENT

Syntax: COMMENT anything you want to say

Specifies a string that is to be included in all text and HTML output files. Useful for identifying information. You can use as many of these as you like. The output files will be more readable if you keep the total line length reasonable, e.g. 75 columns.

Comment lines treat trailing commas as punctuation, not line continuation characters, so multi-line comments must be coded as a separate COMMENT statement for each comment line.

DEBUG

Syntax: DEBUG on|off

Output *lots* of debugging information as the run proceeds. Useful mostly for the developer.

FREQ
LAMBDA

Syntax: FREQ <frequency in Hz>
Syntax: LAMBDA <wavelength in metres>
Specify the frequency or free-space wavelength of the excitation sources (all will be the same). FREQ is an alternative to LAMBDA—use one or the other, but not both.

Examples:

```
FREQ 200*THz  
FREQ c/(1.464*700*nm) /* 700 nm in fused SiO2 */  
LAMBDA 1.55*micron /* Communication wavelength */  
LAMBDA 6.328e-7 /* He-Ne laser */
```

FUNCTION

Define a user math function. User functions can be anything REXX can evaluate, plus elementary functions (sin, cos, tan, asin, acos, atan, atan2, exp, ln), Bessel functions of the first kind of integer order (J_n(x)), complete elliptic integrals, and a few other odds and ends. Once defined, a user function can be used freely in expressions anywhere a literal number or a variable could. Functions may be defined in terms of variables, literals, or other functions, subject to the file parsing rules. When optimization is enabled, functions whose output values depend on variables being optimized over will have their definitions updated on each iteration to reflect the new values of these variables.

If the name of a formal parameter of the function is the same as that of a user variable, the formal parameter will override the user variable inside the function. Otherwise, any variables used by the function will be taken from the global scope. This allows functions to be parameterized by other variables, e.g. scale factors or offsets, which are not in the formal parameter list.

Syntax:

```
FUNCTION name(arg1, arg2,..., argN) = <expression>
```

Examples:

```
FUNCTION myfunc(a,x,q)=exp(a)*atan(q/x)*sqrt(23+q)  
FUNCTION parab(x,y,focus)=(x*x+y*y)/(4*focus)  
FUNCTION cone(x,y)=sqrt(parab(x,y,1))
```

HOSTS

Specifies a file containing TCP/IP information on the hosts available to run the current simulation.

Syntax: HOSTS <hostfile>

where entries in the given file are of the form

```

/* <optional_comment>
<IP> <name> <port> <CPUs> <speed> <RAM MB> <Arch> <OS>
e.g. for a Thinkpad T23 laptop,

/* Kukla--1.13 GHz PIII
127.0.0.1 localhost 1066 1 1.0 768 X86 OS/2 4.5

```

The port number is used for FIDO supervisory control, i.e. if this host is supervising N other hosts, those N connect to the supervisor using this port number. This port can be anything that doesn't conflict with another service on the same cluster, but the unofficial "well-known" port for POEMS is 1066. All hosts can use the same port number.

NB: Hosts are identified by their hostname (as specified by the \$HOSTNAME environment variable, rather than by IP address. This makes it possible to test cluster simulations on a single host by adding multiple names for the same host in the hostfile, and specifying different aliases in different SUBDOMAIN statements in the parameters file. Cluster script fidossl sets the HOSTNAME variable for each fido instance, which means that the host's predefined hostname is not used.

MACDEF

Define a user macro. Macros are parametrized groups of POEMS statements, as opposed to functions, which compute numeric values. Macros are defined and expanded much like C preprocessor macros, except for the MACDEF and MACRO keywords. A macro definition must consist of a single logical line (i.e. if it spans more than one line in the source file, line continuation characters must be used to concatenate them into one logical line). Semicolons must be used to separate individual POEMS statements within the logical line. Both numeric and string values can be passed as macro parameters. Macros do not have local variables other than their pass parameters--other symbolic names will have their normal meaning from the local context at the time of macro expansion. Macro parameter names are dummy variables and will not collide with variables of the same name in the local context--context variables with the same name as the dummy will be inaccessible.

Macros are allowed to contain macros, i.e. the MACRO statement is permitted inside a MACDEF. (This is a powerful and dangerous feature.) Macro recursion is not supported. NB: Since POEMS comments always extend to the end of the current line, any comments inside a macro definition must be at the end of a lexical line, following the line continuation character, as shown below.

Syntax: MACDEF name(arg1, ..., argN) <logical line of code>

Example: This macro plots field files over a fixed volume, with a

given name and field component. The parameters *xsize*, *ysize*, *zsize* are not macro parameters, but will be supplied from the current context when the macro is expanded. Note the string concatenation used to generate the file names.

```
MACDEF FieldAll(kwd, fname)), /* Make I and Q field files
/* with appropriate file names and symbolic limits
FIELD variable=kwd xlo=0 xhi=xmax ylo=0 yhi=yymax ,
zlo=0 zhi=zsize phase=0.0 state=steady file=fname'i' ; ,
FIELD variable=kwd xlo=0 xhi=xsize ylo=0 yhi=ysize ,
zlo=0 zhi=zsize phase=2*atan(1) state=steady /* etc
file=fname'q' ;
```

MACRO

Expand a user macro. Unlike MACDEF, which must be in the GLOBAL group, MACRO can be used anywhere. Macro expansion is typographical, like C macros. String concatenation is performed, as shown in the previous example.

Syntax: MACRO name(arg1, ..., argN);

Example: Using the example from the MACDEF statement,

```
MACRO FieldAll(Ex, Ant01Ex);
```

is equivalent to

```
FIELD variable=Ex xlo=0 xhi=xsize ylo=0 yhi=ysize ,
zlo=0 zhi=zsize phase=0.0 state=steady file=Ant01Exi ; ,
FIELD variable=Ex xlo=0 xhi=xsize ylo=0 yhi=ysize ,
zlo=0 zhi=zsize phase=2*atan(1) state=steady ,
file=Ant01Exq ;
```

POSTPROCESSOR

Tell POEMS which postprocessor executable to use (currently the only choice available is *empost*).

Syntax: POSTPROCESSOR <postprocessor command line>

PRINT

Print output, similar to the REXX SAY statement. Any number of arguments may be supplied, and strings and expressions may occur in any order. Strings inside double quotes will be printed as is, whereas expressions will be evaluated first. PRINT statements are executed each time the input file is parsed, i.e. at the beginning of the run and before each tempest iteration.

Syntax: PRINT <"string"> <expression> ...

RANDOMSEED

Supply a seed to the pseudorandom number generator. Pseudorandom numbers are used only to choose starting points for the optimizer. The points visited by the optimizer for a given problem can be changed by choosing a different seed.

Syntax: RANDOMSEED <expression>

The value of <expression> must be in [0, 999,999,999], and its value is rounded to an integer before use. Thus values must round to different integers for them to be distinct for this purpose.

Example: RANDOMSEED 314159265

SET

Defines a variable and sets its initial value. The expression may contain literals, arithmetic and logical operators, predefined functions, and user functions.

Syntax: SET name = <expression>

Example: set hypot=sqrt(a*a+b*b)

SIMULATOR

Give POEMS a simulator command line to use (no default). For single-host simulations, this is just the simulator executable, but for clusters, it should consist of a command processor, script name, and any additional variables expected by the cluster script. To make this easier, POEMS defines two special variables, as follows.

Name	Expands To
\$h	Name of a file containing a list of all hosts actually used in the simulation, one host per line
\$f	Name of a file containing a list of all files that need to be distributed to the compute hosts for the current simulation.

POEMS constructs the command line by appending an at-sign (@) and the name of a file containing the input parameters expected by FIDO, then passes the resulting command line to the current shell. The script blocks until the simulation returns.

Syntax: SIMULATOR <filename> [<special variable> [<special variable>]]

```
Example: SIMULATOR /usr/local/tempest/tempest
        SIMULATOR rexx d:\poems\fido\fidossh.cmd $h $f
```

TITLE Give a title to the run. This title is printed in each file, and is also used to generate names for the intermediate file and log file, and for the postprocessor orders file. No default.

Syntax: TITLE Waveguide3a

VERBOSE Turn on verbose output. Sometimes helpful in figuring out what's going on. It's sufficiently verbose that it's probably best to redirect it to a file or a pager program such as `less`.

Syntax: VERBOSE on | off

3.1.3. WORLD Group

BOUNDARY Sets boundary conditions for a given axis to be either periodic (PERIODIC) or mirror-symmetrical (SYMMETRY). Only one need be set for each axis; if both are specified, they must be the same, since symmetry applies at both sides in FIDO/TEMPEST. (Deprecated boundary type ILLUM is the same as PERIODIC, and is provided for backward compatibility with earlier versions of POEMS.)

Parameters: xmin xmax ymin ymax zmin zmax

Syntax: BOUNDARY <parameter> PERIODIC | SYMMETRY

SUBDOMAIN Defines a simulation domain. A domain is the portion of the computational world assigned to one thread. Since the current version of tempest runs on a single processor, all domains will be joined together into one big tempest run. FIDO supports multithread simulations, so you don't lose anything by putting them in. Parameter *domain_name* must not contain whitespace. For multiple-host simulations, the SUBDOMAIN keyword takes an optional hostname and base port number, indicating which host this subdomain is to run on, and the name of a supervisor host whose fido instance will control that host. If these parameters are not supplied, the given subdomain runs on the local host. (The supervisor parameter should not be supplied for the host in overall control.) All subdomains running on a given machine must specify the same supervisor, or a runtime error will result. Each

subdomain uses 6 sockets, attached to $port, port+1, \dots, port+5$, all of which must be unique on the given host, though there is nothing to stop different hosts from using the same port numbers. Port numbers must be between 0 and 65535, and it is usually best to use numbers greater than 16384.

Syntax:

SUBDOMAIN *<domain_name>* [host=*<host>* port=*<port>* ,
super=*<super>*]

BASICSTEP

Specify the size of the cubical cells that make up the current subdomain. SI units are suggested, because the scaling between **E** and **H** used internally assumes SI units.

From POEMS's point of view, this number isn't necessarily set in stone: the step size, time step, and domain size can all be controlled by the optimizer or the stepper if desired. For future use, keywords XBASICSTEP, YBASICSTEP, and ZBASICSTEP are also allowed, but since in this release the cells must be cubical, only one of the four may be specified.

Syntax: BASICSTEP *<expression>*

Examples: BASICSTEP lambda/20.5
BASICSTEP 0.1*micron

XRANGE YRANGE ZRANGE

Sets the X, Y, or Z limits of the current domain. The parameters must obey $max_expression > min_expression \geq 0.0$.

Syntax: XRANGE *<min_expression>* *<max_expression>*

3.1.4. MATERIAL Group

DEFINE

Defines the parameters of a material to POEMS.

Parameters: matname type epsReal epsImag muReal muImag *n* *k*
conductivity

Types: dielectric metal PML conductor PEC black magnetic

Type dielectric

Ordinary nonmagnetic material, with $n \geq k$. Most optical

materials are of this type: vacuum, air, glass, and plastics.

Parameters: n , k (real and imaginary refractive index)

Example:

```
DEFINE matname= BlackGlass type=dielectric n=1.52 k=1e-3
```

Type metal

Nonmagnetic dispersive material which may have negative dielectric constant at the frequency of interest, which is equivalent to having $n < k$, e.g. metals in the infrared. Other conductive materials can be modelled as *conductor* type.

There are two ways of specifying a metal: using the n and k values at the centre wavelength of the sources, or using a file produced FITMAT, which fits Debye and Lorentz poles to user-supplied n,k data.

Parameters: n , k , *file*

Example:

```
DEFINE matname= Gold3_lum type=metal n=0.3 k=11.23  
DEFINE matname= Nickel type=metal file=nickel.poles
```

Type conductor

Normal conductor, with $\epsilon = \epsilon_0 + j\sigma/\omega$ Useful for metal waveguide boundaries, at frequencies where the normal conduction model applies (below about 1 THz). In optical simulations, ordinary conductors are primarily useful in simulating loads.

Parameters: *conductivity* (in units of $(\Omega\text{m})^{-1}$)

Example:

```
DEFINE matname=copper type=conductor conductivity=5.8e7
```

Type magnetic

Magnetic material, whose properties are specified by complex μ and ϵ , specified in SI units, *not* the relative μ and ϵ .

Parameters: *epsReal epsImag muReal muImag*

Example:

```
DEFINE name=ferrite type=magnetic epsReal=18,  
epsImag=0 muReal=220 muImag=1.1
```

Type PEC
Type black

Perfect conductor. All fields inside a PEC are zero.
Sometimes useful for saving CPU cycles in regions where the fields are known to be zero, e.g. deep inside metal objects.

Parameters: none

3.1.5. OBJECT Group

Like all dimensions in POEMS, object dimensions are measured from edge to edge, as you would measure with calipers. All dimensions are correctly rounded to the nearest multiple of *basicstep* in the subdomain in which they occur. A cell is included if its centre lies inside the specified region. This can lead to steps at boundaries between subdomains with different *basicstep* values, due to differences in rounding.

When an object's midline is specified in terms of a curve, e.g. FAN, GRATING, CURVE, and 3DCURVE, the end points of the curve are taken to lie at the centre of the outermost plane of blocks on each end. This makes sure that if the local object axis is highly inclined, the outermost blocks don't disappear due to their centres falling outside the region, as they would if the endpoint were taken to lie in the outer face. Sufficiently inclined objects may still become discontinuous.

BLOCK

Adds a rectangular prism (like a shoe box) of material *matname*, covering the region $xlo \leq x \leq xhi$, $ylo \leq y \leq yhi$, $zlo \leq z \leq zhi$. Like everything else in POEMS, these dimensions will be rounded to the nearest cell boundaries.

Parameters: *matname* *xlo* *xhi* *ylo* *yhi* *zlo* *zhi*

Example:

```
block matname = AirPML xlo=0 xhi=8*dx ylo = 0 yhi=ymax ,  
      zlo=0 zhi=zmax
```

FAN

Adds a fan shaped object such as a dielectric waveguide horn or taper. A fan is specified by a choice of curve and rectangular end faces 1 and 2, defined by their diagonal points $((xlo1, ylo1, zlo1), (xhi1, yhi1, zhi1))$ and $((xlo2, ylo2, zlo2), (xhi2, yhi2, zhi2))$. These must be parallel and lie in a coordinate plane. Intervening planes are defined by the choice of taper and the taper parameter, which is a scale factor for the domain of the curve—increasing the parameter causes the taper to be sharper, and decreasing it makes the taper more gradual. Allowed tapers are LINEAR, EXPONENTIAL, and ERF. Exponential tapers are useful for converting guided waves to free space, and ERF tapers are useful for converting between different-sized waveguides without strong back reflections from either end.

If the two end faces are the same size and shape, all the curve shapes are equivalent. The end faces can be offset laterally, so that a fan statement can build a diagonal line.

Parameters: *matname taper taperpar xlo1 xhi1 ylo1 yhi1 zlo1 zhi1 xlo2 xhi2 ylo2 yhi2 zlo2 zhi2*

GRATING

Adds a planar grating with lines of rectangular cross-section. The line width and phase of the grating are arbitrary, and specified with user functions of the coordinate variables. Thus the grating can have its properties altered in a very general way during optimization, e.g. having its Seidel aberration coefficients controlled to optimize a coupling efficiency. Gratings are implemented by evaluating *phasefunc* at a grid of points in the plane, and constructing a phase contour map. Each grating line is generated by following the contours at integral multiples of 2π , and centring a rectangular block of total width defined by *widthfunc* on the contour line, so that the grating's diffractive strength is also a function of position. Making the width too large or too small will result in the spaces or lines disappearing, which is one way of making a grating with a non-rectangular boundary. The default spacing between phase points is two cells, but this can cause the script to run very slowly, so it can be overridden with the optional parameter *interval*. The *interval* parameter determines the grid on which the phase and width functions are sampled, which influences the accuracy of the contour maps and hence the grating placement.

Parameters: *matname orientation xlo xhi ylo yhi zlo zhi widthfunc phasefunc [interval]*

HOLLOWBOX

Adds a hollow box (rectangular shell) of specified outer dimensions and thickness. It is implemented by dividing up the rectangular box into 26 smaller blocks: 6 for the faces, 12 for the edges, and 8 for the corners. This primitive is especially useful for using PMLs to isolate a region from its surroundings. When used with PMLs, the outer dimensions must be the same as those of the simulation domain. The PML absorption directions will be the outward normals for the flat faces, outward-directed face diagonals for the edges, and outward-directed body diagonals for the corners.

Parameters: *mattype xlo xhi ylo yhi zlo zhi thickness*

TILEDPLANE

Adds a tiled plane (thick rectangular sheet) of specified outer dimensions. It is implemented by dividing up the rectangular sheet into 9 smaller blocks: 1 for the face, 4 for the edges, and 4 for the

corners. This primitive is especially useful with plane wave illumination, where we want PMLs only on the front and back faces of the region. The wide face of the object must touch one of the faces of the simulation domain. The PML absorption directions will be the outward normals for the face, outward-directed face diagonals for the edges, and outward-directed body diagonals for the corners. If for some reason you call this with an ordinary material, it just adds a regular block.

Parameters: *mattype xlo xhi ylo yhi zlo zhi*

BLOB

Adds a generalized planar object whose shape is determined by a user-defined function, $f(u,v)$ where u and v are the coordinates lying in the plane. Cells whose centres lie inside regions where $f(u,v) > 0$ will be filled with material.

Parameters: *matname xlo xhi ylo yhi zlo zhi outlinefunc*

Example:

```
BLOB matname=gold xlo=0 xhi=xsize/2 ylo=0, yhi=ysize/2,
zlo=zsize/2-step zhi=zsize/2 orientation=XY,
outlinefunc= x*x/(xsize*xsize)+y*y/(ysize*ysize)-xsize*ysize/4
```

CURVE

Adds a curved object whose path is expressed by a user-defined planar curve, $(u(s), v(s))$, and whose outline is swept out by the region $f(q,r) > 0$ as the parameter s advances from slo to shi . Variable q is the normal to the plane, and r is parallel to $q \times (du/ds, dv/ds)$. This statement can be used to make very general curved objects of variable cross-section. Due to the uniform rectangular grid employed by FIDO/TEMPEST, curved objects will look somewhat different depending on the direction of their axes at any given place. POEMS attempts to minimize this effect.

Parameters: *mattype slo shi perp hfunc vfunc*
Not implemented in this release.

3DCURVE

Like CURVE, but the axial curve is allowed to be nonplanar. This statement is more complicated to use, due to the noncommuting property of 3-D rotations—in specifying the functions u, v, w , it's easy to get mixed up by the way they rotate around.

Parameters:
Not implemented in this release.

CYLINDER Adds an object whose axis is a straight line, but whose outline is an arbitrary function of the perpendicular coordinate.

Parameters: *matttype axis lo hi maxradius insidefunc*
Not implemented in this release.

3.1.6. SOURCE Group

GAUSSIAN Produces a TEM00 Gaussian beam, centred at (x, y, z) , with $1/e^2$ diameter *width*, whose axis is specified by the vector (kx, ky, kz) , which can have any length. The **E** field is linearly polarized, with components $(mag*Ex, mag*Ey, mag*Ez)$. Circular or elliptical polarizations can be synthesized by using two with different phases. In this release, *orientation* must be *XY*, because tempest can't handle plane waves whose source locus isn't perpendicular to *z*. Also in this release, the focus of the beam is at the plane of excitation, i.e. *xfocus*, *yfocus*, and *zfocus* don't do anything yet. This will be fixed in the next release.

Parameters: *x y z width kx ky kz Ex Ey Ez mag phase n k orientation*

AIRY Similar to Gaussian, but with a uniform pupil function, giving rise to an Airy pattern $(J_1(x)/x)$ at the focus.

Parameters:*Not implemented in this release.*

PUPILFUNCTION Similar to GAUSSIAN, but allowing a general user function $f(u,v)$ to be used as the pupil function. GAUSSIAN is already implemented on top of this function, but it hasn't been exposed for this release.

Parameters:*Not implemented in this release.*

PLANE Adds a linearly polarized plane wave source. The position parameters *x, y, z* specify the point at which the plane wave has the given phase.

Parameters: *x y z kx ky kz Ex Ey Ez mag phase orientation*

MODEFILE Adds a mode file source. Mode files are lists of **E** field components vs position, produced by the MODEFILE output statement in a

previous POEMS run, allowing the output of one run to become the input of another. This is especially useful in waveguide problems such as the one in Section 3.6.3. The coordinates specified are the centre of the mode array, which is in general offset from the origin.

Parameters: *file x y z orientation*

POINT

Adds a linearly-polarized point electric dipole source at the given position. Point sources whose polarization is not x , y , or z (diagonally, circularly, or elliptically polarized) can be implemented as two or three point sources with appropriately chosen magnitude and phase.

Parameters: *x y z Ex Ey Ez mag phase*

3.1.7. COMMAND Group

COMPUTE

Supply control parameters to FIDO/TEMPEST. Parameters *mincycles* and *maxcycles* control the number of cycles of the excitation frequency that each simulation may use; setting the lower limit occasionally helps in avoiding spurious early convergence, and the upper puts a bound on the amount of run time. Adjusting *maxcycles* to a small number such as 1-3 is useful when setting up a run, because you can get rapid feedback on whether your geometry is correct, and whether your PMLs are likely to cause problems. Slightly higher values are useful in the early stages of an optimization, where extreme accuracy in penalty function evaluation is not needed, but run time is a serious concern. See the TEMPEST documentation for more details. Start with a value of 0.01 or 0.001 for *reltolerance*, and don't set *mincycles* or *timestep* until you really need to.

Parameters: *reltolerance maxcycles mincycles timestep*

3.1.8. OUTPUT Group

INDEXN

Tells tempest to produce a field file containing the real part of the refractive index at each point. This is useful to show which material is where in the simulation domain.

Parameters: *xlo xhi ylo yhi zlo zhi file*

DECIMATE Specify that only 1 out of *decimation* points is to be kept in each axis.

Parameters: *xdecimation ydecimation zdecimation*
Not implemented in this release.

FIELD Tells tempest to produce a field file containing the specified field variable at each point.

Parameters: *variable xlo xhi ylo yhi zlo zhi x y z file*
phase state

3.1.9. POSTPROCESS Group

CAD Produces a .ctxt (2-D) or 2-D or 3-D DXF file of the entire simulation domain, suitable for importation into a CAD program, e.g. for numerically controlled machining, photomask generation, or documentation. In a 3-D DXF file, each block in the simulation domain becomes a set of six 3DFACE entities. In a .ctxt or 2-D DXF file, each block is projected onto two rectangles, representing the upper and lower faces in the given orientation. POEMS DXF files are known to import properly into several CAD packages including Autodesk Inventor. Note that length unit used in the DXF file is metres unless otherwise specified--since many CAD packages use a fixed number of decimals, this may cause confusion when the sample domain is very small (e.g. in optical waveguides). Note that the orientation is ignored in a 3-D file.

Parameters: *file filetype dimensions orientation units xcut ycut zcut space*

Examples: CAD file=ofile3.dxf filetype=dxfile dimensions=3
CAD file=ofile2.dxf filetype=dxfile dimensions=2,
units=micron orientation=ZX
CAD file=ofile.ctxt filetype=ctxt units=micron,
orientation=XY

The CAD statement also permits defining a line (for 2-D) or a plane (3-D), parallel to a coordinate axis, at which extra space is to be inserted, for use with shadow mask fabrication. For instance, for a bridge width of 1.5 μm , you can add the following to the CAD statement:

CAD file=mask.ctxt filetype=ctxt units=micron orientation=XY,
xcut=1.8*um space=1.5*um

WEBPAGE	<p>Produces a static HTML page containing all the relevant run parameters, output bitmaps, and links to other files such as DXF and Vis5D.</p> <p>Parameters: <i>file</i></p>
FARFIELD	<p>Computes the far-field limit of the fields at a given plane, specified as $xlo \leq x \leq xhi$, $ylo \leq y \leq yhi$, $zlo \leq z \leq zhi$. Because this range must specify a plane, the upper and lower limits of one axis must be the same to within $basicstep/2$. The coordinates will be correctly rounded to integral numbers of cells. We assume that the fields propagate through an infinite half-space of refractive index n (which obviously must be lossless). The refractive index is taken to be the real part of the index in the actual model at the centre of the supplied plane, $\{xmin \leq x \leq xmax, ymin \leq y \leq ymax, zmin \leq z \leq zmax\}$. The parameter <i>direction</i> can be <i>up</i> or <i>down</i> depending on which direction we're interested in. Internally, POEMS uses both E and H field information to separate out the incoming and outgoing fields, so the computed far field spectrum with <i>direction=down</i> can be very different from that with <i>direction=up</i>. At present this statement produces bitmaps of <i>s</i>- and <i>p</i>-polarized field amplitude and phase at each point (u, v) in the pupil plane.</p> <p>Currently the FARFIELD statement can be applied only on planes of uniform granularity, i.e. <i>basicstep</i> must be the same in all subdomains cut by the given plane.</p> <p>Parameters: <i>name xlo xhi ylo yhi zlo zhi file direction</i></p> <hr style="width: 20%; margin: 10px auto;"/>
FLUX	<p>Computes an integral of the Poynting vector through the given surface, in the <i>inward</i> direction. This isn't quite the same as using the INTEGRAL statement directly, because it adds the ability to specify an interior point. A positive flux is going in the direction towards from $(xInside, yInside, zInside)$.</p> <p>Parameters: <i>name xinside yinside zinside xlo xhi ylo yhi zlo zhi</i></p> <hr style="width: 20%; margin: 10px auto;"/>
INTEGRAL	<p>Computes a volume or surface integral of a given field function by summing all the blocks lying in the specified region. If the region has a nonzero thickness (after rounding to the nearest multiple of <i>BASICSTEP</i>), it's normalized as a volume integral; if two have a nonzero thickness, it's normalized as a surface integral of the broad face of the region; and if only one dimension has a nonzero size, it's normalized as a line integral. Specify the integrand field or postprocess quantity as <i>variable</i>, and make sure there's a matching</p>

statement that generates an array of the given quantity, because it won't be done automatically.

Parameters: *name variable xlo xhi ylo yhi zlo zhi*

LIST

Stores a list of field values as ASCII floating point numbers, suitable for reading into a spreadsheet or a plotting program such as GNUPLOT. The row-column arrangement of a 3-dimensional list file depends on the value of *orientation*. Coordinate axes are always taken in cyclic order, with the leftmost position in the first line corresponding to the minimum of all coordinates; in a 2-D list in XY orientation (perpendicular to Z), columns correspond to the x coordinate and rows to the y coordinate. In YZ orientation, y goes across columns and z goes down rows, and in ZX orientation, z goes across columns and x down rows. (ZY is the same as YZ, and XZ is the same as ZX--cyclic order is always preserved.) In a three-dimensional list, the perpendicular variable is most slowly varying. For example, in XY orientation, a 3-D list file with M planes (Z) N rows (Y) and P columns (X) would have list N rows for the first plane, another N for the second plane, and so on. The file format is as follows, with integer indices i , j , and k corresponding to coordinates x , y , and z .

Parameters: *variable file orientation xlo xhi ylo yhi zlo zhi phase indexn*

```
kmin kmax jmin jmax imin imax
lambda indexN dz dy dx
coordinates arrangement
(real, imag) (real, imag) (real, imag)....
(real, imag) (real, imag) (real, imag)....
...
(real, imag) (real, imag) (real, imag)....
```

The value of *coordinates* can be *polar* or *rectangular*, and *arrangement* can be *normal* or *FFT*.

MODEFILE

A variation on the LIST command, for producing 2-D list files of E_x , E_y , and E_z . The appropriate file name additions are supplied automatically.

Parameters: *file orientation xlo xhi ylo yhi zlo zhi phase indexn*

MODEMATCH

Computes the far-field pattern of the simulated fields, taken from the given plane, and compares it with the analytically computed pupil function requested. It returns the normalized overlap integral of the two across the (u, v) plane. This is a complicated way of saying that MODEMATCH returns the coupling efficiency from the simulated plane to a receiver whose sensitivity pattern matches the

given pupil function, e.g. a fibre. Known pupil functions at present are GAUSSIAN, AIRY, FLATTOP, and BESSJ0. The refractive index is taken to be the real part of the index at the centre of the given plane, as in the FARFIELD order.

Parameters: *name xinside yinside zinside function exi exq eyi eqz ezi ezq xlo xhi ylo yhi zlo zhi xfocus yfocus zfocus xNA yNA zNA NA direction*

MOVIE

Produces an animated one-cycle GIF of N frames, over the given plane.

Parameters: *file orientation xlo xhi ylo yhi zlo zhi dt frames variable*

Not implemented in this release.

MOVIE3D

Produces an interactive Vis5D animation of up to 9 field variables. Setting *frames=1* (the default) produces an instantaneous snapshot, which you can explore interactively, which is most suitable for time-averaged and time-independent quantities such as refractive index and power dissipation density. To see the complex-valued fields change, set *frames* to a positive integer N . This produces an N -frame 3-D animation covering one cycle of the excitation, the i^{th} frame corresponding to a phase of i/N (counting from zero).

Due to a limitation of Vis5D, the size of the space cannot exceed 100 cells in each direction. If you call for a region larger than this, it will be decimated automatically by the smallest integer factor that allows it to fit into 100 pixels. Up to 100 frames of animation may be specified, and any POEMS variables are allowed. There is no provision for setting phase angles, as the animation will cover a full cycle.

Parameters: *frames variable1 variable2 variable3 variable4 variable5 variable6 variable7 variable8 variable9 xlo xhi ylo yhi zlo zhi file*

AMPLEX, AMPLEY
AMPLEZ, AMPLHX
AMPLHY, AMPLHZ
PHASEEX, PHASEEY
PHASEEZ, PHASEHX
PHASEHY, PHASEHZ
POYNTINGX,
POYNTINGY
POYNTINGZ

These statements compute the amplitude and phase of **E** and **H** fields, and the time-averaged Poynting vector. The output is a tempest-style field file suitable for further processing. In the current release, each field component desired must be specified manually before anything else is done to it, because otherwise the array won't

exist when it's needed. There's a wish list item to make the postprocessor smarter about file handling and memory utilization.
Parameters: *xlo xhi ylo yhi zlo zhi x y z file*

DISSIPATION Power dissipation density , $-\nabla \cdot \mathbf{S}$ (the Poynting vector). Normalized to the volume of one cell, so that volume integrals of this quantity give the correct power dissipation inside the region.
Parameters: *name xlo xhi ylo yhi zlo zhi file xinside yinside zinside*

SLICE Produce a bitmap of *variable* over the given domain, with control over the colour palette and the scale. Variables are always taken in cyclic order, i.e. *x, y, z, x, y, z,...*, which preserves the right-handedness of the coordinate system. The positive sense of the perpendicular axis is always out of the screen towards the viewer. (Technical detail: The Windows bitmap format has the pixel index starting at (0, 0) in the upper left corner of the screen, so that the column index corresponds to position correctly but the row index is inverted. POEMS flips the row index internally, so that the screen coordinates are correct for the simulation geometry--positive *X* is to the right, positive *Y* is up.)

Perpendicular	Orientation	Increasing to right	Increasing upwards	Increasing out of the screen
Z	XY	X	Y	Z
X	YZ	Y	Z	X
Y	ZX	Z	X	Y

Table 1: Axis orientations for SLICE bitmaps

Formats: The available formats are BMP, GIF, JPEG, and PNG, determined by the file extension supplied (i.e. if the filename ends in '.png', it's a PNG file). If the filename has no extension, it's assumed to be a BMP file. POEMS generates BMP files internally, and relies on the 'convert' program from the ImageMagick package to do the translation. If you want something other than BMP files, you'll need to install ImageMagick.

Parameters: *orientation variable phase xlo xhi ylo yhi zlo zhi file curve palette palfile*

Choices of *palette* are *grey, saturation, flame, bluered, and custom*. Grey scale palettes are simple and unexciting, but clear. Saturation palettes keep the luminance level nearly constant (except that value

0 is black), but vary the hue from yellow to a saturated blue. Flame palettes start out black, go through dark grey, dull red, bright red, orange, yellow, white, and blue-white, mimicking the colour of a hot object. The bluered palette is suitable for bipolar values such as **E** and **H** fields; it forces white to be at value 0, with increasing red saturation for positive values and increasing blue saturation for negative values. A custom palette requires a palette file *palfile*.

These are text files with the form

```
level  R      G      B<newline>
level  R      G      B<newline>
```

...

where *level*, *R*, *G*, and *B* are integers between 0 and 255, inclusive. The *level* column represents the signal level, and *R*, *G*, and *B* are the usual RGB colour components.

Choices of *curve* are *linear*, *gamma*, *logarithmic*, and *MuLaw*. The linear curve spaces the palette entries out evenly, with the maximum value in the picture being 255 and the minimum, 0. The gamma curve first takes the square root of the absolute value of the variable, restores the sign, and spaces out the result evenly. This accentuates details in the weaker-field regions, without losing definition in the higher-field regions. The logarithmic curve accentuates weak fields very strongly, which is very useful in some cases but tends to lose detail where the field is strong. The mathematical logarithm is modified to avoid floating-point runtime errors: it actually computes

$$\text{index} = \frac{255}{\ln(10^4)} \ln(1+10^4|x|)$$

where *x* has been normalized by the largest element in absolute value. The MuLaw curve uses the CCITT μ -255 codec curve, which is linear for values near the positive and negative limits, but puts the levels close together near zero. While this resembles a bipolar version of the logarithmic curve, it actually combines many of the advantages of the linear and logarithmic curves, since not all detail is lost at the highest values.

Syntax: SLICE file = <fname> xlo=<value> xhi=<value> ,
ylo=<value> yhi=<value> zlo=<value> zhi=<value> ,
phase=<value> variable = Ex

Example:

```
SLICE file = /var/tmp/v1aExIsourceZ.png xlo=0 xhi=xsize ,
ylo=0 yhi=ysize zlo=1.5*um zhi=1.5*um phase=0 ,
variable = Ex
```


3.1.10. OPTIMIZE Group

- VARIABLES** Tell the optimizer which variables to optimize over. These must have been defined in a SET statement.
Syntax: VARIABLES *var1 ... varN*
- GUESS** Supplies a guess to the optimizer. A guess is a point in the N -dimensional parameter space. For an N -dimensional optimization run, $N+1$ guesses are required, each supplying all N variable values. The first guess is supplied by the SET statements where the optimization variables are defined, but additional ones can be supplied by GUESS statements. If more than $N+1$ guesses are supplied, guess $N+2$ overwrites guess 1, and so on. You can supply a pre-computed value for the penalty or merit function, by including
merit=<expression> or
penalty=<expression>
in the GUESS statement. While this can save valuable time, it must be used with care, because POEMS will trust the value you give it. If the value is stale or otherwise incorrect, this will interfere with the optimization. If the supplied value is unrealistically poor, the optimizer will incorrectly tend to move away from it, good, whereas if it is too good, the optimizer may get stuck in that vicinity, trying to improve a value that is better than any available computed value.

Syntax: GUESS <var1> = <expression> <var2> =, <expression>....
[penalty = <expression>]
- LIMIT** Expression that has to be true for a valid point. Often it's better to change the simulation parameters to build the limits in--e.g. replacing width with abs(width) or using more complicated functions of absolute values. LIMIT is better for more complicated functions, e.g. forcing the corners of some structure to lie within an arbitrary boundary. *Not implemented in this release.*
Syntax: LIMIT <Boolean expression>

Example: LIMIT length*width*height ≤ MaxVolume
- STORE** Specify a file to store the simplex for each iteration, along with the corresponding penalty function values. These are in the right format to be cut and pasted back into the parameters file to restart an iteration without losing previous work.
Syntax: STORE <filename>
- PENALTY**
MERIT Defines an expression on which the optimizer is to work. Merit functions are maximized, and penalty functions are minimized.

Only one of the two can be specified. In order for the optimizer to do anything useful, the optimization expression must depend on each of the optimization variables. *Note that any variable assignment involving quantities computed by the postprocessor will not be updated at postprocessing time.* That means that the dependence on this iteration's results must be included explicitly in <expression>. For example, if the iteration is minimizing the loss of a coupler, you might specify `MERIT 10*log10(efficiency)`, which will work fine, but you can't use `SET dBloss=-10*log10(efficiency)` up in the GLOBAL section and then specify `PENALTY dBloss` in the OPTIMIZE section--if you do, you'll wind up with stale data. This can sometimes be useful, but not usually.

Syntax: `MERIT <expression>` OR
`PENALTY <expression>`

Example: `PENALTY 100*(OutputFlux/InputFlux - 1)`

PARAMETERS Defines quantities to guide the optimizer. The number of function evaluations (i.e., temperst runs) per restart will not exceed *iterations*, convergence will be declared when the penalty function value in all the simplex points is within *tolerance* of the best value, and the optimizer will be restarted *restarts* times after converging or running out of iterations.

Parameters: *iterations tolerance restarts*

Example:

`PARAMETERS iterations=35 tolerance = 1e-3 restarts = 2`

3.1.11. SCHEDULE Group

Stepping is not implemented in this release.

VARIABLES Tell the stepper which variables to step. These must have been defined in a SET statement.

Syntax: `VARIABLES var1 ... varN`

RANGE Tell the stepper how far, and at what intervals, to step variable *varname*.

Parameters: *varname lo hi stepsize*

STORE Supply a file name for the stepper's results.

Syntax: `STORE <filename>`

3.2. The Computational Domain

3.2.1. Symmetry

Many technologically useful devices have a periodic structure. The periodic boundary conditions assumed by POEMS are a natural fit to such structures, which can be simulated very efficiently. This happens automatically if you leave out the absorbers on the sides facing the periodic direction. Almost as many devices exhibit symmetry planes. A domain with a symmetry plane, illuminated symmetrically, will exhibit completely symmetric fields, so no extra information is generated by simulating both halves. Splitting the domain in half down the symmetry boundary can save half the run time and half the memory, while preserving the same level of detail in the results. Domains with more than one symmetry boundary save even more (though this begins to restrict the illumination more severely). POEMS (via support built into TEMPEST, FIDO, and EMPOST) supports symmetrical simulations.

It's important to visualize the symmetry arrangement correctly, and it isn't what one might first think: A symmetry plane actually slices right through the centre of the outermost elements. Say the X axis has 100 cells, numbered 0-99. Normally, periodic boundary conditions apply, and cell 0's neighbours are 1 and 99. If X is a symmetry direction, cell 0's neighbour is cell 1 *on both sides*. That is, a 1-block wide layer lying against the symmetry axis has the same effect as a 1-block layer in the middle of the unfolded domain. A 2-block layer against the symmetry axis is equivalent to a 3-block layer down the middle of the unfolded domain.

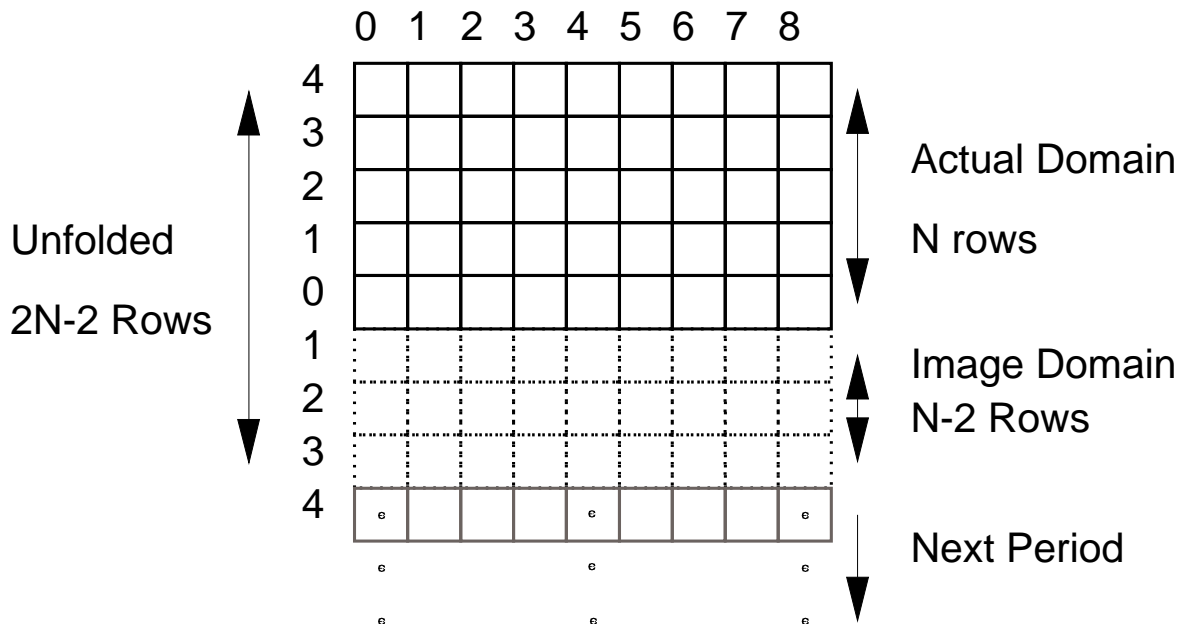


Figure 2.2 Geometry of a symmetry boundary with $N=5$

Another way of stating this is that if a symmetry plane is like a mirror, then we'd expect that a block pushed right up against it would look like two blocks. With POEMS, we have to think of it as pushing a block up to *and halfway through* the mirror, so it still looks like one block. A one-block layer at the symmetry axis unfolds to one block, a two-block layer unfolds to three blocks, and in general N blocks unfold to $2N-1$ blocks. Furthermore, you can't have symmetry just on one side--if X is a symmetry direction, both the x_{min} and x_{max} faces are symmetry planes.

We're typically using symmetry to improve the tradeoff between the block size and the number of blocks required--in other words, trying to get the finest level of detail we can afford. Implementing symmetry the other way, so that a one-block layer at the symmetry axis unfolds to 2 blocks, means that there is no way to get 1-block details at the symmetry axis. At least one symmetry plane is usually right in the middle of the region of interest, so this reduces the value of symmetry in saving computation. Thus this seemingly odd choice is in fact superior. However, it does have significant consequences for postprocessing, e.g. you can't just compute the flux over half the area and double it, because that will count the blocks on the symmetry axis twice. The postprocessor code handles this by halving the weights of blocks lying against the symmetry axis. The halving happens separately for each axis, so that if there are two symmetry directions, the symmetry face blocks have weight 0.5 and the ones down the intersections of the symmetry planes have weight 0.25. Thus the integrals and fluxes computed by POEMS *can* be simply multiplied by a factor of 2 per symmetry axis to yield the correct result for the unfolded domain.

We commonly want to run a symmetric simulation once in the full domain and once in the halved or quartered domain, to make sure everything comes out the same. To do this in POEMS, remember that if the folded dimension has N cells, its invisible other side has $N-2$ cells. (The end cells are special.) Thus the full domain will be $2N-2$ cells wide. If you run the full domain first, you must give it an even number of cells in the symmetry direction.

Symmetrical domains of course must have symmetrical illumination. The TEMPEST approach is to forbid such domains to have off-axis plane waves applied to them. The other approach would have been to double each plane wave source, so that their result was symmetric, which would be preferable. Since the `xxBEAM` source statements are implemented as sums of plane waves, we can't use them on symmetric domains. Use mode sources instead. It isn't difficult to make a mode source out of a beam source: just run a three-cycle simulation of the desired beam shining into an empty domain, and make the mode file in a plane near the source.

3.3. OBJECTS

Everything in the simulation domain is made up of cubical blocks arranged in a simple cubic lattice, so there's a limit to how subtle we can be in specifying curves and angles. Nonetheless, POEMS provides a wide array of object types that can be parameterized in powerful ways, as discussed in the command reference.

3.3.1. Perfectly-Matched Layers

PMLs are fictitious anisotropic materials that (usually) absorb whatever radiation falls on them from pre-specified directions. They work extremely well when they work at all, which isn't 100% of the time.

PMLs don't absorb in every direction, so different parts of the domain need different PMLs. This is such a pain that POEMS makes a valiant effort to automate it for you. It works as follows:

- (i) If a PML touches a boundary, it will be set to absorb energy falling on that boundary from the centre of the domain.
- (ii) If a PML touches more than one boundary, rule(i) will be applied to all of them.
- (iii) PMLs that touch both the upper and lower boundary of an axis will be automatically split in half down the middle so that (ii) can be applied to both halves.
- (iv) PMLs must touch at least one boundary.

A solid block of PML filling the whole space is thus split into 8 blocks, each absorbing diagonally outwards. You can manipulate these rules to customize PMLs for your use. One common desire is to have PMLs covering the whole outer skin of the domain, with the faces, edges, and vertices all absorbing in different directions (26 blocks altogether). This is fiddly, so the HOLLOWBOX statement was added to POEMS to take care of it. HOLLOWBOX is implemented by splitting the box up and letting the automatic PML handler apply rules (i)-(iv) to each block. The TILEDPLANE statement does the same for planar PML regions.

POEMS has heuristics for generating reasonable PMLs automatically, but these are not always reliable. Signs of trouble are checkerboard patterns, fields apparently originating in PML regions, floating-point runtime errors, and the generation of NaNs. This will happen every time if you put a material boundary near a PML, or if you have sources inside it. A reasonably safe rule is to put a PML no closer than 1 wavelength from a material boundary or a source. Plane waves and PMLs are a difficult combination in TEMPEST.

3.4. MATERIALS

Materials used in POEMS are treated as isotropic, linear and time-invariant. tempest supports photoresist materials and has limited support for anisotropic ones, so if this becomes important it can be added in a future release.

3.5. SOURCES

POEMS has a rich array of sources, and when the PUPILFUNCTION order becomes available, it will be able to generate essentially any monochromatic propagating field.

3.5.1. POINT SOURCES

In continuous-space electrodynamics, static point sources are mathematically very simple. In electrodynamics, there are no monopole sources, since a monopole field is purely radial, whereas electromagnetic waves are purely transverse. The simplest physical

source is therefore an oscillating dipole. FDTD codes model a dipole as a single-cell source whose E field oscillates in time, just as you'd expect. Unfortunately, only linearly-polarized sources are supported because only a single point source can occupy a cell. This restriction is not essential, and may be removed in a future release.

The other problematical thing about point sources is normalization. One might expect that a point source of strength 1 would produce an E field at the source of 1, but that isn't how TEMPEST's point sources are normalized. This and other normalization problems (of which there are a few) will be fixed in a future release. The problem doesn't affect power ratios and efficiencies, which is what we usually care about.



Figure 2.3 TEMPEST divergence due to source inside PML. This box is 3.2 wavelengths across.

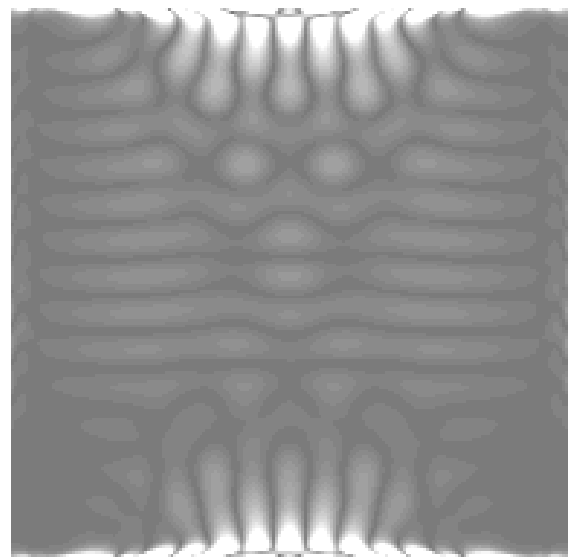


Figure 2.4: Side view of the domain of Figure 2.3.

3.5.2. PLANE WAVES

Because of the periodic boundary conditions assumed in the simulation geometry, POEMS has a lot in common with FFT programs, e.g. point N is the same as point 0 , and only waves whose spatial frequencies are integral multiples of $1/(N \Delta x)$ can be used. Since $\Delta x \ll \lambda$, and memory is limited, in practice only a few dozen or at most a few hundred plane wave components are necessary to synthesize any desired pupil function. The down side is that the resulting beams are spatially periodic, and their side lobes will leak into adjacent domains.

A plane wave source produces radiation in only one direction, and does not interfere with other field components crossing it. Plane waves don't work well with fully isolated domains, i.e. those with PMLs on all surfaces. It's useful to put PML boundaries downstream of the illumination, but when using plane waves, some imaginary absorbing material boundary is often better behind the illumination surface.

FDTD codes use periodic boundary conditions. Fields leaving through one side will magically reappear coming from the other, which may lead to divergences if you don't put in absorbers of some sort. Sloppy absorber design may lead to fields travelling many, many periods, leading to anomalously slow convergence. Figure 2.4 is an example of a divergence due to PMLs used with plane wave sources. Note that the divergence looks electrostatic—the wavelength is far too short to propagate. This is a good clue that something is wrong.

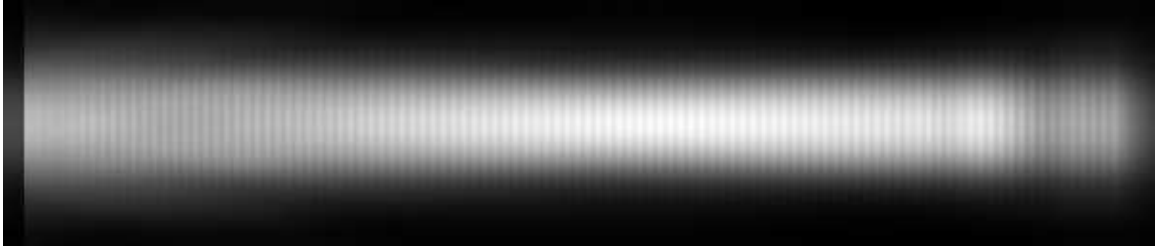


Figure 2.5: 120- μm long doped silica waveguide, excited with a circular Gaussian beam of diameter equal to its core width. A black glass region is at each end (waveguide1c.par).

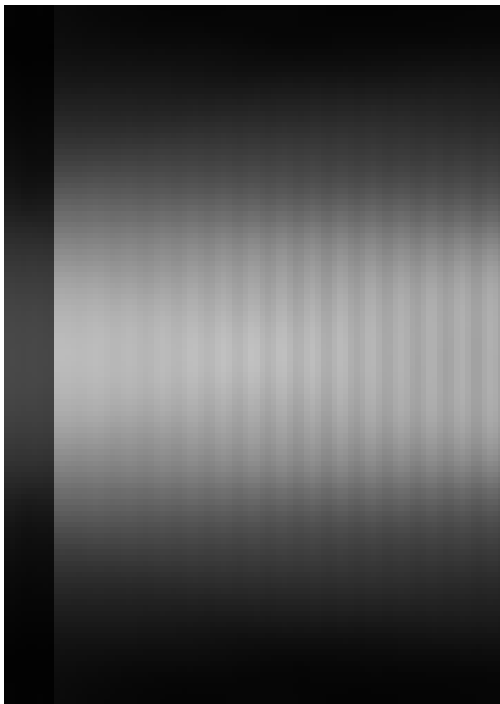


Figure 2.6 Detail of the launch end. Note the unidirectional character of the Gaussian beam source, and the weak reflected field leaking back through the illumination plane after travelling about 110 μm .

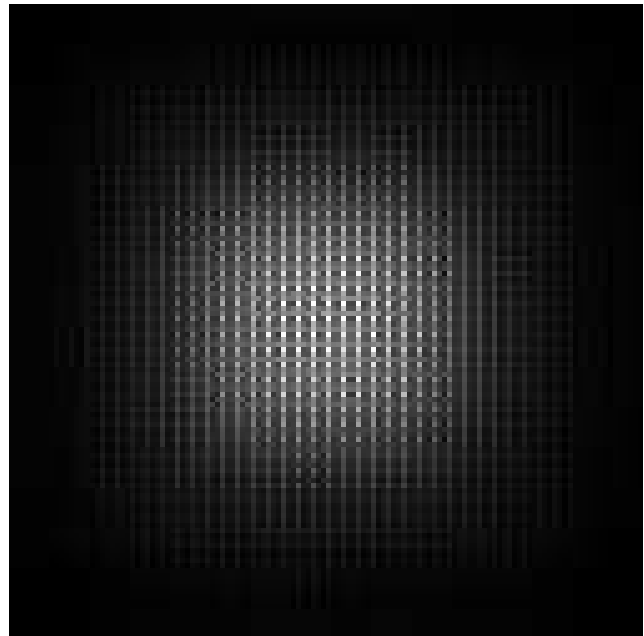


Figure 2.7: Mode source for a 5- μm doped-silica waveguide ($\Delta n=0.02$). This slice is taken at the plane of the sources, showing the strong nonuniformity.

3.5.3. BEAM SOURCES

Beam sources are made up of sums of plane waves. They share the unidirectional property of plane sources. This is sometimes useful in waveguide simulations, because you can bounce the wave off a mirror at the far end and look at the field coming backwards through the plane wave surface. Since all of this field has travelled twice the length of the waveguide, it can make a good approximation of the waveguide mode. This is where the mode file used in 3.5.4 came from.

3.5.4. MODEFILE SOURCES

Modefile sources are actually implemented as arrays of point sources. TEMPEST can't eat its own output, and furthermore it has a very limited number of point sources available (as few as 100 in the vanilla 6.0 release), so it's hard to do a good job of representing a waveguide mode.

From the modefile generated by the postprocessor (via the MODEFILE statement of the POSTPROCESS group), POEMS computes $N \times N$ boxcar averages of the E field, and generates point_sources spaced N cells apart in each perpendicular direction. The decimation factor N is chosen to be at most $0.3\lambda/(n dx)$, so that the evanescent field dies out by a factor of 1000 in at most $0.8 \lambda/n$.

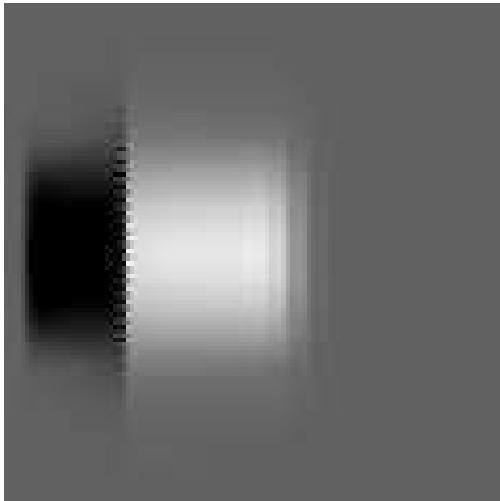


Figure 2.9 Plot of the Z (axial) component of the Poynting vector in the simulation of Fig. 3.3, after 5 cycles. A PML absorbs the $-Z$ wave (dark), leaving a clean $+Z$ wave. Leading ripples are an artifact of the quadrature-field calculation.

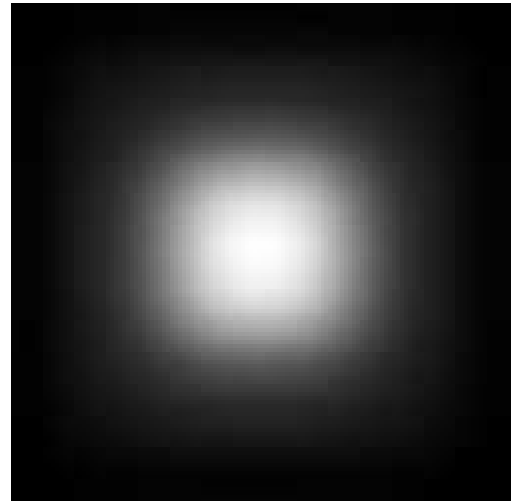


Figure 2.8: The same mode source as in Figure 2.7, but showing a slice taken $0.8 \mu\text{m}$ downstream. By this point the mode has evened out completely because the strongly evanescent ripple components have died away.

3.6. Optimization

There are powerful techniques for finding the minimum of smooth functions in a few function evaluations, an important consideration when the merit function requires minutes to hours for each iteration. There are robust techniques for finding the minimum of functions that are noisy, discontinuous, or otherwise badly behaved, which our simulation results tend to be. There are intelligent techniques that have a good prospect of finding a global minimum of a function whose geometry is poorly known, which ours usually are. Unfortunately for us, the intersection of those three groups is the null set.

Because 3-D electromagnetic simulations tend to be slow, noisy, and to produce merit function values full of multidimensional cliffs and canyons, we have to be realistic about what optimization can give us. Given reasonable computing resources and time to run, POEMS will take your initial guess and improve it automatically. It will explore the immediate neighbourhood, and if after a couple of restarts it converges back to the same place, you can be reasonably sure that there isn't a significantly better design in the immediate vicinity. If you run out of time and have to stop it somewhere, the best point in the current simplex is always the best point evaluated so far.

The POEMS optimizer is more like an automatic design-of-experiments machine than an elegant numerical thoroughbred, but it is a reasonable choice for the problem at hand. Work is underway to find a better one, with emphasis on response-surface optimizers, where the merit function values go into a statistically-fitted parameterized surface, and the minimum of that surface is used as the next guess. Pruning the set of points used and choosing the right class of surfaces are the tricky parts.

The Nelder-Mead downhill simplex method does not always converge to the minimum, even of smooth functions. When this happens, the reason is that the simplex has closed up on itself in some dimension, so that its hypervolume becomes anomalously small and the search space becomes defective. For this reason, once it has converged, it's usually a good idea to restart it using the best point and a few random values. The POEMS optimizer does this if requested. On average, Nelder-Mead seems to be the most efficient direct-search algorithm available for the problem at hand—its occasional failure to converge without restarting is more than balanced by its use of only 1 or 2 function evaluations per step (except for shrink steps).

3.6.1. Merit Functions

In order for this to do anything useful, you have to specify a merit function (or penalty function for pessimists) that depends on the simulation output. POEMS provides enough flexibility to optimize nearly anything technologically relevant, except that it has no direct support for nonlinear optics (which it may in the future). A few examples are useful.

a. Waveguide Loss Loss in a waveguide device can be calculated from the flux integrals in the input and output planes:

```
POSTPROCESS  
...
```

```

FLUX      name=PwrIn xlo=0 xmax=10*um ylo=0 ymax=12*um z=zSource+1*um,
          zinside = zSource+2*um xinside = xsize/2 yinside=ysize/2
FLUX      name=PwrIn xlo=0 xmax=10*um ylo=0 ymax=12*um z=zOutput,
          zinside = zSource+2*um xinside = xsize/2 yinside=ysize/2
END
OPTIMIZE
PENALTY   (PwrIn-PwrOut)/PwrIn
...
END

```

b. Mode matching between a device output and a Gaussian beam focused at $(X, Y, Z) = (5, 5, 20) \mu\text{m}$, propagating along the +Z direction, with the integral being carried out from 0 to 10 μm in X and Y, in the plane $Z=3 \mu\text{m}$.

```

POSTPROCESS
...
MODEMATCH name=GaussianWaistUp ,
          xinside= xsize/2,
          yinside=ysize/2,
          zinside=zsize/2,
          function=CircularGaussian,
          exi=1 exq=0,
          eyi=0 eyq=0,
          ezi=0 ezq=0,
          xlo=0,
          xhi=10*um,
          ylo=0,
          yhi=10*um
          zlo=3*um,
          zhi=3*um,
          xfocus=5*um,
          yfocus=5*um,
          zfocus=20*um,
          NA= 0.35,
          direction=up,
          file=somerandomfile.dat

```

c. Phase uniformity across a plane

There are several ways to do this. If you want the beam to go in a particular direction, you can compute the mode match with a very low NA uniform beam; or if you want the wavefront to be planar but don't care exactly which direction it's going, you can compute integrals of the phase in small regions and compute the deviation from the best straight line between them. This is more messy than it is difficult.

d. RMS Voltage across terminal points

If the terminals are separated in X, this could be (assuming they're much less than a wavelength apart):

```

INTEGRAL name=Vreal variable=Ex phase=0 xlo xhi ylo yhi zlo zhi
INTEGRAL name=Vimag variable=Ex phase=pi/2 xlo xhi ylo yhi zlo zhi
MERIT     sqrt(Vreal*Vreal+Vimag*Vimag)

```

e. Power dissipation in a load

dissipation

f. Reflection from an interface

FLUX behind plane source/(FLUX in front+FLUX behind)

3.6.2. Worked Example: Optimizing a V Antenna

Thermodynamics dictates that the total power an antenna can deliver to its terminals when excited by isotropic thermal radiation cannot be larger than kT per hertz (otherwise you could put a hot resistor at the feed point, point the antenna at a colder object, and have the resistor get hotter spontaneously). Thus there is a 1:1 trade-off between antenna gain and solid angle: the product of the effective intercepted area of an antenna and the equivalent width of its angular pattern in steradians cannot exceed $\lambda^2/2$. The angular pattern of an ordinary dipole has a solid angle of π steradians, so its gain is only 2. We can do better, for narrower angular patterns, by raking the ends of the dipole forwards into a V shape. This optimization run is adjusting the width and rake of the arms, and the feed point impedance, to maximize the power dissipated in the load.

The parameters file used in this simulation is in Appendix A, along with the simplex file, showing the progress of the optimization.

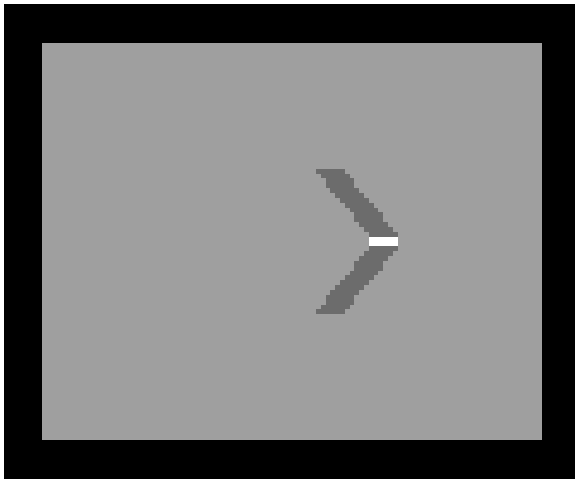


Figure 2.10 Optimized V antenna: refractive index distribution

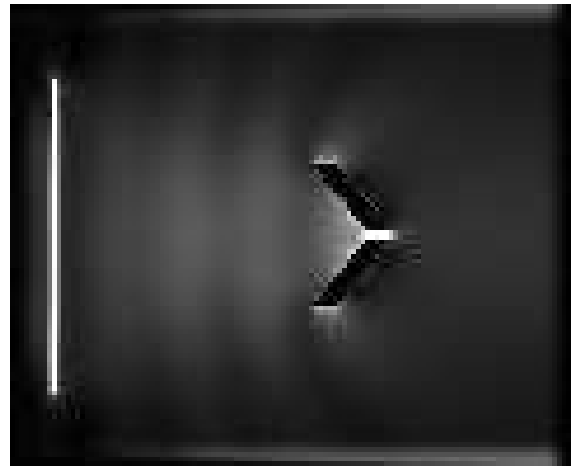


Figure 2.11 Optimized V antenna: E field amplitude

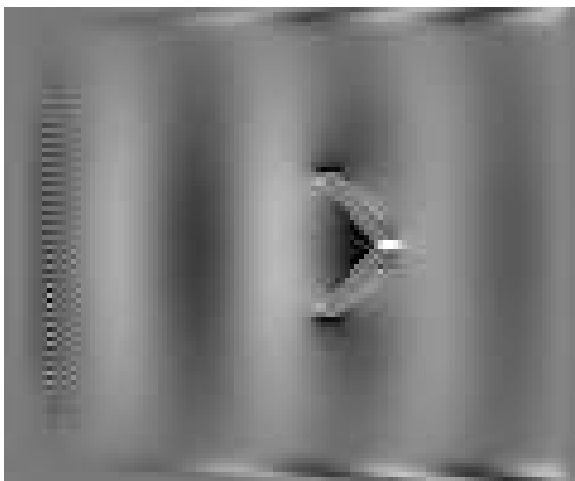


Figure 2.12 Optimized V-antenna: Quadrature component of the E field

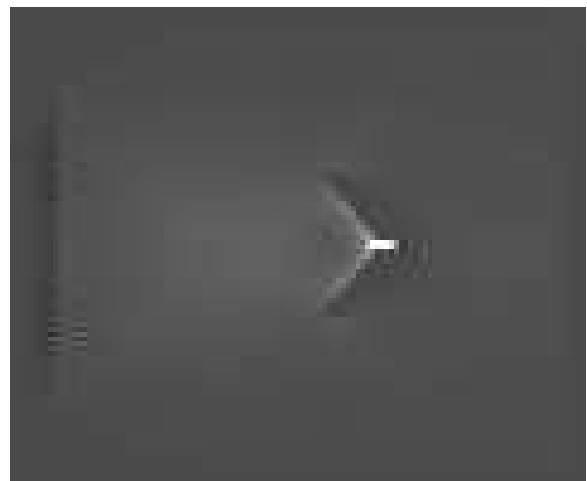


Figure 2.13 Optimized V antenna: Z component of the Poynting vector

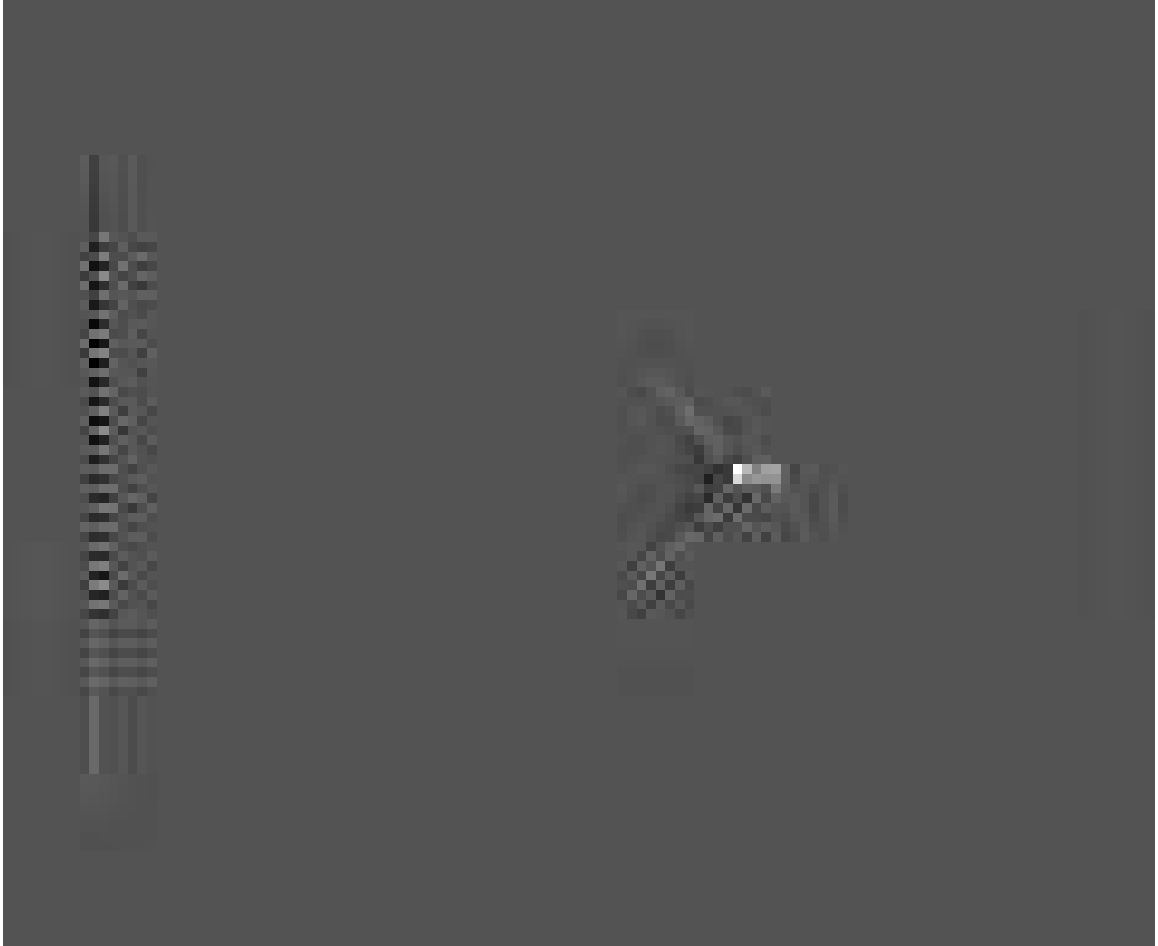


Figure 2.14 Optimized V antenna: power dissipation density. Note the very large maximum at the feed point.

3.6.3. Worked Example: Doped Silica Waveguide Mode

In simulating waveguide devices, the first requirement is to have a source that produces a pure waveguide mode. Sometimes a Gaussian beam is sufficient, but often it is not, especially when small reflections or phase shifts are important. Analytical solutions are not often available for the modes of real dielectric guides, so a numerical procedure is of more general use. A good one is to illuminate a long section of model waveguide with some approximate mode, and let the leaky components lose themselves. In this example, we build a 60- μm long doped silica core guide with a 5- μm square core and an index difference of 0.02. Absorbing material with the same index as the core lines the edges of the region, to suck up the leaky field. In this simulation, we take advantage of the unidirectional character of the plane wave sources to gain double the propagation distance: we let the wave bounce off the far end of the guide and return through the source plane. We take a modefile output just behind the source plane, where the wave has had 110 μm or so to be purified. If desired, the treatment can be repeated by running this

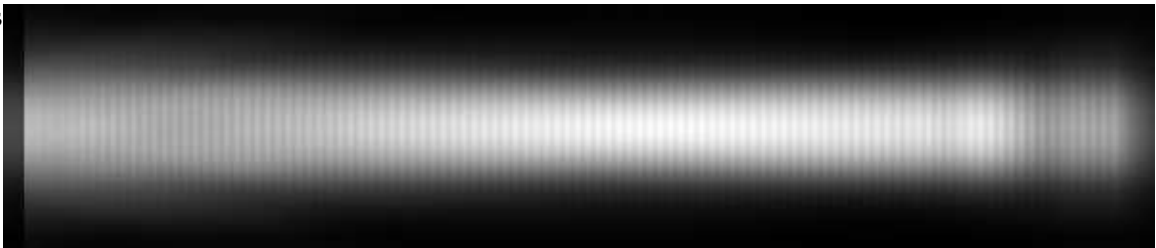


Figure 2.15: 60- μm long doped silica waveguide, excited with a circular Gaussian beam of diameter equal to its core width. A black glass region is at each end (waveguide1c.par).

3.6.4. Worked Example: Glass Ridge Waveguide to Free Space Coupler

3.7. Predefined Constants

The following constants and units are predefined for convenience. SI units are used throughout. These names are reserved and cannot be redefined.

m	=	1.0
km	=	1000*m
cm	=	m/100
mm	=	cm/10
um	=	mm/1000
micron	=	um
nm	=	um/1000
angstrom	=	nm/10
s	=	1
ms	=	s/1000
us	=	ms/1000
ns	=	us/1000
ps	=	ns/1000

fs	=	ps/1000
as	=	fs/1000
kHz	=	1000
MHz	=	1000*kHz
GHz	=	1000*MHz
THz	=	1000*GHz
EHz	=	1000*THz
pi	=	3.141592653589793
twopi	=	2*pi
deg	=	pi/180
degree	=	deg
rad	=	1.0
cycle	=	twopi
c	=	299792458
mu0	=	4*pi*1e-7
eps0	=	1/(c*c*mu0)

3.7.1. Reserved Names

Defining variables with the same name as a function (user or built-in) is legal but not a good idea. If you do it, the variable will override the function in all subsequent statements. Because of the potential for confusion, it is forbidden to define a user function with the same name as a built-in function. Built-in variables have the same precedence as user variables (i.e. the highest) so defining functions with these names is forbidden, because they could never be evaluated.

List of all reserved variable names:

freq lambda omega x y z k u v w
 Hz kHz MHz GHz THz EHz cycle deg degree rad
 pi twopi km m cm mm um micron nm angstrom
 s ms us ns ps fs as c mu0 eps0 tol penalty merit

List of reserved function names:

ABS MAX MIN FORMAT TRUNC ROUND ROOT1D MINIMIZE INTEGRAL SQRT POW SIN COS
 TAN ATAN ATAN2 ASIN ACOS LN EXP ERF ELINTK ELINTK1 BOWTIEZ0 AU_N AU_K SI_N
 SI_K ASI_N ASI_K PSI_N PSI_K NI_N NI_K CU_N CU_K SIN_N SIN_K SIO2_N SIO2_K
 CONFINE

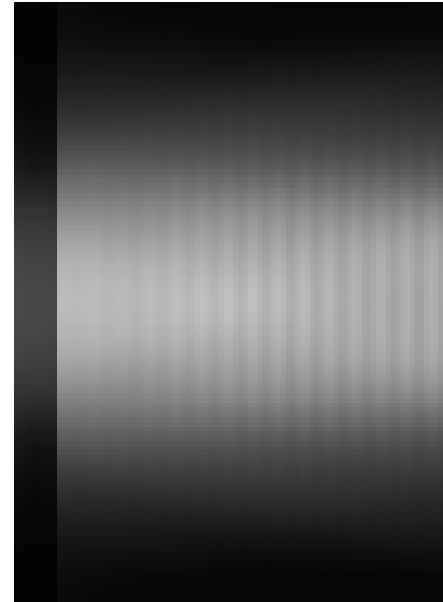


Figure 2.16 Detail of the launch end. Note the unidirectional character of the Gaussian beam source, and the weak reflected field leaking back through the illumination plane after travelling about 110 μm .

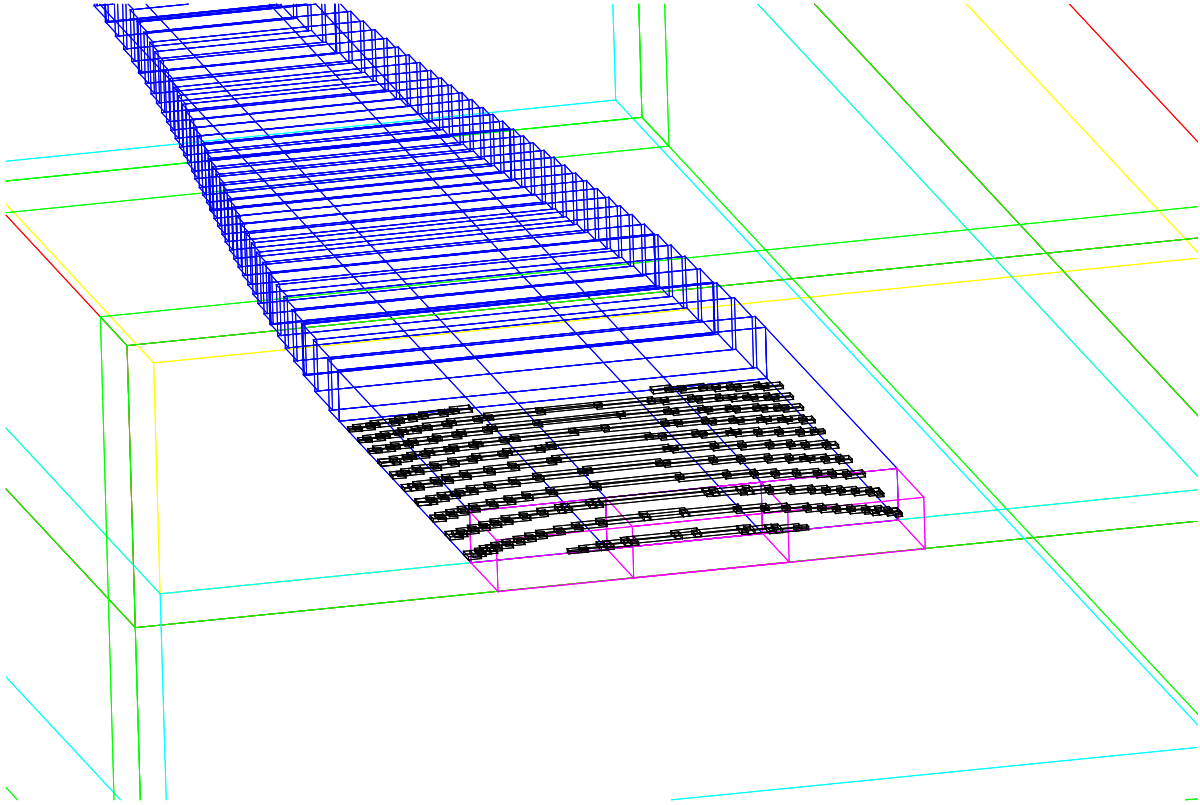


Figure 2.17: Detail of the CAD output for one iteration of the glass ridge waveguide to free space coupler.

3.8. Predefined Mathematical Functions

3.8.1. Arithmetic Operators

The user-facing part of POEMS is implemented in REXX, so the arithmetic and logical operators and many of the simpler mathematical functions are native REXX constructs, and have REXX syntax and semantics.

+	(addition)
-	(subtraction)
*	(multiplication)
/	(division (real numbers))
//	remainder (remainder has the same sign as the numerator)
%	integer division

a. Differences from C/C++

C programmers should beware of a few things: the '%' operator, which in C and C++ is the modulus (remainder) operator, is used for ordinary integer division in REXX (and so in POEMS). Use the double-slash ('//') operator for modulus.

Also, REXX does not distinguish between integer and floating-point numbers. It isn't that it uses floats for everything—internally REXX uses arbitrary-precision arithmetic.

3.8.2. Logical Operators

> (greater than)

< (less than)

≥ (greater or equal)

≤ (less or equal)

<> (unequal)

\ (not)

3.8.3. ABS

Returns the absolute value of *<expression>*.

Syntax: abs(*<expression>*)

3.8.4. ACOS

Returns the radian angle in $[0, 2\pi)$ whose cosine is *<expression>*. Expressions whose values are outside $[-1,1]$ will result in run-time errors.

Syntax: acos(*<expression>*)

3.8.5. ACOSH

Returns the argument in $[0, \infty)$ whose hyperbolic cosine is *<expression>*. Attempting to take the hyperbolic arc-cosine of a number less than 1 will result in a run-time error.

Syntax: acosh(*<expression>*)

3.8.6. ASIN

Returns the radian angle in $(-\pi, \pi]$ whose sine is *<expression>*.

Syntax: asin(*<expression>*)

3.8.7. ASINH

Returns the argument whose hyperbolic sine is *<expression>*. ASINH(*x*) is defined for all values of *x*.

Syntax: asinh(*<expression>*)

3.8.8. ATAN

Returns the radian angle in $(-\pi/2, \pi/2)$ whose tangent is *<expression>*.

Syntax: atan(*<expression>*)

3.8.9. ATANH

Returns the argument whose hyperbolic tangent is *<expression>*. Attempting to take the hyperbolic arctangent of a number outside $(-1,1)$ will result in a run-time error.

3.8.10. ATAN2

Returns the radian angle in $(-\pi, \pi]$ whose tangent is $\langle sine_expression \rangle / \langle cosine_expression \rangle$, and which is in the correct quadrant.

Syntax: atan2($\langle sine_expression \rangle$, $\langle cosine_expression \rangle$)

3.8.11. CEIL

Returns the smallest integer that is at least as large as the argument (i.e., round towards $+\infty$). More useful than integer truncation, since it Does The Right Thing with negative arguments.

Syntax: ceil($\langle expression \rangle$)

3.8.12. COS

Returns the cosine of radian argument $\langle expression \rangle$.

Syntax: cos($\langle expression \rangle$)

3.8.13. ELINTK

Complete elliptic integral of the first kind. Used in computation of bowtie antenna feed impedances. See Abramowitz & Stegun

Syntax: elintk($\langle expression \rangle$)

3.8.14. ELINTK1

Complete elliptic integral of the first kind, of complementary argument. Used in computation of bowtie antenna feed impedances

Syntax: elintk1($\langle expression \rangle$)

3.8.15. ERF

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-x^2}$$

Error function,

. erf($-\infty$) = -1.0, erf(0)=0 & erf(∞) = 1.0.

Syntax: erf($\langle expression \rangle$)

3.8.16. EXP

Exponential e^x .

Syntax: exp($\langle expression \rangle$)

3.8.17. FLOOR

Returns the largest integer that is not greater than the argument (i.e., round towards $-\infty$). More useful than integer truncation, since it Does The Right Thing with negative arguments.

Syntax: floor($\langle expression \rangle$)

3.8.18. FORMAT

REXX formatted print function. See a REXX manual for full details.

Syntax: format($\langle expression \rangle$, nleft, nright, exptol, nexp)

nleft=number of places to the left of the decimal

nright=number of places to the right of the decimal
 exptol=trigger for exponential notation
 nexp=number of places in the exponent

FORMAT('12345.73',,,,2,2)	→	'1.234573E+04'
FORMAT('12345.73',,,,3,,0)	→	'1.235E+4'
FORMAT('1.234573',,,,3,,0)	→	'1.235'
FORMAT('12345.73',,,,3,6)	→	'12345.73'
FORMAT('1234567e5',,,,3,0)	→	'123456700000.000'

3.8.19. INTEGRAL

Computes the 1-dimensional integral of user function f from x_{lo} to x_{hi} . Just a stub at present.

Syntax: INTEGRAL(f , x_{lo} , x_{hi})

3.8.20. LN

Natural logarithm.

Syntax: ln(*<expression>*)

3.8.21. MAX

Returns the most positive of its arguments.

Syntax: max(*<expression>*, *<expression>*, ..., *<expression>*)

3.8.22. MIN

Returns the most negative of its arguments.

Syntax: min(*<expression>*, *<expression>*, ..., *<expression>*)

3.8.23. CONFINE

Constrains the given value to be not less than the minimum and not more than the maximum values (if given). Confine works by reflecting the given value off the limits, so that the confined value is a continuous function of all its inputs. The graph of CONFINE(x , upper, lower) vs x is a triangle wave; that of CONFINE(x ,,lower), a V shape; and that of CONFINE(x ,upper) an upside-down V shape.

Syntax: confine(x , upper)
 confine(x , , lower)
 confine(x , upper, lower)

3.8.24. MINIMIZE

Finds the value of x at which user function $f(x)$ is minimized, within the limits $x_{lo} \leq x \leq x_{hi}$. Currently just a stub.

Syntax: MINIMIZE(f , x_{lo} , x_{hi})

3.8.25. POW

Computes $\text{arg}1^{\text{arg}2}$. No error checking is performed for raising 0 to a negative power or a negative number to a fractional power.

Syntax: Pow(*<expression>*, *<exponent_expression>*)

3.8.26. RANDOM

Interface to the REXX pseudorandom number generator. Returns a pseudorandom integer from *<min_expression>* to *<max_expression>*, inclusive, and optionally re-seeds the generator with *<seed_expression>*, evaluated and rounded to an integer. Default values : *<min_expression* ≥ 0, *<max_expression* ≥ 100,000. One restriction is that the difference between the maximum and minimum limits must not exceed 100,000.

Syntax: Random(*<min_expression>*, *<max_expression>*, *<seed_expression>*)

3.8.27. ROOT1D

Finds a root of the given expression. Guess1 and guess2 must have opposite signs. Currently just a stub.

Syntax: Root1D(*expression*, *unknown*, *guess1* *guess2*, *min* *max*)

3.8.28. ROUND

Rounds *expression* to the specified number of decimal places, or to the nearest integer. Numbers exactly halfway between steps are rounded up.

Syntax: Round(*<expression>*[, *<places>*])

3.8.29. SIGN

Signum function. Returns +1, 0, -1 for *<expression>* >0, =0, and <0.

Syntax: Sign(*<expression>*)

3.8.30. SIN

Returns the sine of radian angle *<expression>*.

Syntax: sin(*<expression>*)

3.8.31. SQRT

Returns the square root of a non-negative number.

Syntax: Sqrt(*<expression>*)

3.8.32. TAN

Returns the tangent of radian angle *<expression>*.

Syntax: tan(*<expression>*)

3.8.33. TRUNC

Evaluate *<expression>* and truncate (round towards zero).

Syntax: trunc(*<expression>*)

3.9. Analytical Pupil Functions

POEMS knows about the following analytical pupil functions:

GAUSSIAN	TEM ₀₀ Gaussian beam
AIRY	Uniform circular disc pupil function, resulting in an Airy function at the focus $E=2J_1(krNA)/(krNA)$, the jinc function.
FLATTOP	Pupil function is a jinc times a circularly-symmetrical Hamming window, resulting in a focused beam with a flat top and smoothly sloping sides. The actual pupil function is $E(u,v)=2(0.54+0.46\cos(\pi\rho))J_1(6\pi\rho)/(6\pi\rho)$, where $\rho=(u^2+v^2)^{1/2}/NA$.

3.10. Material Parameter Functions

These functions are linearly-interpolated values generated from tables in Palik & Ghosh, *Optical Constants of Materials*. They are parameterized by wavelength in metres, and are in general much more precise than they are accurate. It isn't that the measurements are bad, or that the interpolation is too crude, it's just that they describe measurements on particular films prepared with extreme care. Most optical devices are made with films whose density is not 100%, whose structure is columnar rather than amorphous or crystalline, and whose stoichiometry is far from perfect, so these values may not accurately predict the refractive indices of your actual materials. "Optical constants of thin films" is an oxymoron, so don't take these values too seriously.

These discrepancies are perhaps most serious in the case of fused silica. Commercial fused silica usually contains some OH groups due to the silane flame deposition technique used to make fused silica boules. These OH groups are harmless in the visible, but out at 2 μm and beyond (especially near 2.7 μm) they may make the material almost opaque, even though the function SiO2_K() predicts low absorption there. (The best infrared-grade fused silica is indeed transparent in this region.) Of course, given the memory and processor speed constraints of current machines (say 2 GB and 3 GHz), the opacity of OH hasn't much of a chance to be felt within the bounds of a FDTD simulation.

Au_N(lambda), Au_K(lambda)	Gold	0.5636 — 5.166 μm
Ni_N(lambda), Ni_K(lambda)	Nickel	0.5636 — 5.166 μm
Cu_N(lambda), Cu_K(lambda)	Copper	0.5636 — 5.166 μm
Si_N(lambda), Si_K(lambda)	Crystalline silicon	0.5636 — 5.166 μm
aSi_N(lambda), aSi_K(lambda)	amorphous silicon	1.03 — 1.52 μm
SiN_N(lambda), SiN_K(lambda)	Silicon Nitride	under construction
SiO2_N(lambda), SiO2_K(lambda)	Fused silica	0.5636 — 5.166 μm
Corning1737_N(lambda), μm"	Corning 1737 glass	0.4360 - 1.541 μm"