[COMPLEX RELUCTANCE OF INHOMOGENEOUS EULER-CAUCHY TUBULAR FERRITES TAKING INTO ACCOUNT FREQUENCY-DEPENDENT COMPLEX PERMEABILITY J. A. Brand˜ao Faria]

|  |  |  |
| --- | --- | --- |
|  | Electric | Magnetic |
| Skin Effect inclusion |  |  |
| Losses |  |  |
| Axial fields. Euler- Cauchy Equation | P is homogeneity parameter. | P is homogeneity parameter. |
| Stored Energy |  |  |
| Circuit laws |  |  |

[A Matrix Approach for the Evaluation of the Internal Impedance of Multilayered Cylindrical Structures]

|  |  |  |
| --- | --- | --- |
|  | Electric | Magnetic |
|  | k is complex wave number |  |

[FORMULATION OF MULTIWIRE MAGNETIC TRANSMISSION-LINE THEORY]

|  |  |  |
| --- | --- | --- |
|  | Electric | Magnetic |
|  |  | is the flux rate. U is magnetic voltage. |

[Matrix theory of wave propagation in hybrid electric/magnetic multiwire transmission line systems]

|  |  |  |
| --- | --- | --- |
|  | Electric | Magnetic |
|  |  |  |

[MEEP]

|  |  |  |
| --- | --- | --- |
|  | Electric | Magnetic |
|  | **P** is polarization density | **M** is polarization density. |
| Lorentzian susceptibility | Sum of harmonic resonances plus a term form frequency independent electric conductivity. Sigma-n couples the polarization to the driving field, omega-n is the angular frequency of precession, gamma-n is a damping factor.  Term containing bn (bias vector) is responsible for gyrotropy. Precession occurs around bn unit vector. It represents the angular frequency of precession induced by the external field.  For ferromagnetic materials, gyromagnetic saturated dipole Landau-Lifshitz-Gilbert equation is used |  |
| Susceptibility tensor for gyrotropic Lorentzian model | For ferromagnetic materials, gyromagnetic saturated dipole Landau-Lifshitz-Gilbert model, |  |
| Instantaneous isotropic Pockels and Kerr Non-linearity | can be changed by the E field. Chi-2 sum is the Pockels effect. Chi-3 sum is the Kerr effect |  |
| SI units Weber Convention  Ampere’s Law |  |  |
| Faraday’s Law of Induction |  |  |
| Gauss’s Laws |  |  |
| Lorentz Force Equation |  |  |

ABB Transformers Handbook

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| Inrush currents |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[High-Performance Planar Isolated Current Sensor for Power Electronics Applications Luca Dalessandro, Student Member, IEEE, Nicolas Karrer, Member, IEEE, and Johann W. Kolar, Senior Member, IEEE]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

[Designing Wide-band Transformers for HF and VHF Power Amplifiers By Chris Trask, N7ZWY]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[A Simplified Analysis of the Broadband Transmission Line Transformer By Jerry Sevick]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

[Pulse Transformer Datasheet UE-TS\*121A]

|  |  |  |
| --- | --- | --- |
|  |  |  |

[Wideband Transformers: An Intuitive Approach to Models, Characterization and Design By Chris Trask]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

[**(WO2013089574) MAGNETIC TRANSMISSION LINE DEVICE FOR TERAHERTZ INTEGRATED CIRCUITS**]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

[Wikipedia Transformer Types]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

[Wikipedia Power dividers and directional couplers]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[Permeability Engineering Toolbox]

|  |
| --- |
|  |

[Permalloy Wikipedia]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

[Maxwell’s Equations Wikipedia]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

[Magnetometer Wikipedia]

|  |  |  |
| --- | --- | --- |
| Hall Effect |  |  |
|  |  |  |
|  |  |  |

[Epstein Frame Wikipedia]

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |

[MEEP Documentation]

|  |  |
| --- | --- |
| MEEP\_H | **Susceptibility**  polarizability vector P = chi(omega) W (where W = E or H)  **Lorentzian Susceptibility**  a Lorentzian susceptibility  \chi(\omega) = sigma \* omega\_0^2 / (\omega\_0^2 - \omega^2 - i\gamma \omega)  If no\_omega\_0\_denominator is true, then we omit the omega\_0^2 factor in the  denominator to obtain a Drude model.  **class noisy\_lorentzian\_susceptibility**  like a Lorentzian susceptibility, but the polarization equation  includes white noise with a specified amplitude \*/  **multilevel\_susceptibility**  int L; // number of atom levels  int T; // number of optical transitions  realnum \*Gamma; // LxL matrix of relaxation rates Gamma[i\*L+j] from i -> j  realnum \*N0; // L initial populations  realnum \*alpha; // LxT matrix of transition coefficients 1/omega  realnum \*omega; // T transition frequencies  realnum \*gamma; // T optical loss rates  realnum \*sigmat; // 5\*T transition-specific sigma-diagonal factors  **h5file**  // h5file.cpp: HDF5 file I/O. Most users, if they use this  // class at all, will only use the constructor to open the file, and  // will otherwise use the fields::output\_hdf5 functions.  **material\_function**  /\* This class is used to compute position-dependent material properties  like the dielectric function, permeability (mu), polarizability sigma,  nonlinearities, et cetera.  **simple\_material\_function**  simple\_material\_function(double (\*func)(const vec &)) { f = func; }  **structure\_chunk**  bool condinv\_stale; // true if condinv needs to be recomputed  double \*sig[5], \*kap[5], \*siginv[5]; // conductivity array for uPML  int sigsize[5]; // conductivity array size  grid\_volume gv; // integer grid\_volume that could be bigger than non-overlapping v below  volume v;  susceptibility \*chiP[NUM\_FIELD\_TYPES]; // only E\_stuff and H\_stuff are used  int refcount; // reference count of objects using this structure\_chunk  **class boundary\_region {**  boundary\_region\_kind kind;  double thickness, Rasymptotic, mean\_stretch;  pml\_profile\_func pml\_profile;  void \*pml\_profile\_data;  double pml\_profile\_integral, pml\_profile\_integral\_u;  direction d;  boundary\_side side;  boundary\_region \*next;  **PML**  boundary\_region pml(double thickness, direction d, boundary\_side side, double Rasymptotic = 1e-15,double mean\_stretch = 1.0);  **structure**  structure\_chunk \*\*chunks;  int num\_chunks;  bool shared\_chunks; // whether modifications to chunks will be visible to fields objects  grid\_volume gv, user\_volume;  double a, Courant, dt; // res. a, Courant num., and timestep dt=Courant/a  volume v;  symmetry S;  const char \*outdir;  grid\_volume \*effort\_volumes;  double \*effort;  int num\_effort\_volumes;  **src\_time**  // Time-dependence of a current source, intended to be overridden by  // subclasses. current() and dipole() are be related by  // current = d(dipole)/dt (or rather, the finite-difference equivalent).  **gaussian\_src\_time**  // Gaussian-envelope source with given frequency, width, peak-time, cutoff  **continuous\_src\_time**  // Continuous (CW) source with (optional) slow turn-on and/or turn-off.  **custom\_src\_time**  void \*data;  double start\_time, end\_time;  **monitor\_point**  vec loc;  double t;  std::complex<double> f[NUM\_FIELD\_COMPONENTS];  monitor\_point \*next;  // dft.cpp  // this should normally only be created with fields::add\_dft  **dft\_chunk**  double omega\_min, domega;  int Nomega;  component c; // component to DFT (possibly transformed by symmetry)  size\_t N; // number of spatial points (on epsilon grid)  std::complex<realnum> \*dft; // N x Nomega array of DFT values.  class dft\_chunk \*next\_in\_chunk; // per-fields\_chunk list of DFT chunks  class dft\_chunk \*next\_in\_dft; // next for this particular DFT vol./component  std::complex<double> stored\_weight;  bool include\_dV\_and\_interp\_weights;  bool sqrt\_dV\_and\_interp\_weights;  std::complex<double> extra\_weight;  fields\_chunk \*fc;  ivec is, ie;  vec s0, s1, e0, e1;  double dV0, dV1;  bool empty\_dim[5]; // which directions correspond to empty dimensions in original volume  std::complex<double> scale; // scale factor \* phase from shift and symmetry  ivec shift;  symmetry S;  int sn;  std::complex<realnum> \*dft\_phase;  ptrdiff\_t avg1, avg2; // index offsets for average to get epsilon grid  int vc; // component descriptor from the original volume  **dft\_flux**  // dft.cpp (normally created with fields::add\_dft\_flux)  double freq\_min, dfreq;  int Nfreq;  dft\_chunk \*E, \*H;  component cE, cH;  volume where;  direction normal\_direction;  bool use\_symmetry;  **dft\_energy**  // dft.cpp (normally created with fields::add\_dft\_energy)  double freq\_min, dfreq;  int Nfreq;  dft\_chunk \*E, \*H, \*D, \*B;  volume where;  **dft\_force**  // stress.cpp (normally created with fields::add\_dft\_force)  double freq\_min, dfreq;  int Nfreq;  dft\_chunk \*offdiag1, \*offdiag2, \*diag;  volume where;  **dft\_near2far**  // near2far.cpp (normally created with fields::add\_dft\_near2far)  double freq\_min, dfreq;  int Nfreq;  dft\_chunk \*F;  double eps, mu;  volume where;  direction periodic\_d[2];  int periodic\_n[2];  double periodic\_k[2], period[2];  **dft\_ldos**  /\* Class to compute local-density-of-states spectra: the power spectrum  P(omega) of the work done by the sources. Specialized to handle only  the case where all sources have the same time dependence, which greatly  simplifies things because then we can do the spatial integral of E\*J  \*first\* and then do the Fourier transform, eliminating the need to  store the Fourier transform per point or per current. \*/  std::complex<realnum> \*Fdft; // Nomega array of field \* J\*(x) DFT values  std::complex<realnum> \*Jdft; // Nomega array of J(t) DFT values  double Jsum; // sum of |J| over all points  public:  double omega\_min, domega;  int Nomega;  **dft\_fields**  // dft.cpp (normally created with fields::add\_dft\_fields)  double freq\_min, dfreq;  int Nfreq;  dft\_chunk \*chunks;  volume where;  **fields\_chunk**  fields\_chunk(structure\_chunk \*, const char \*outdir, double m, double beta,bool zero\_fields\_near\_cylorigin);  **boundary\_condition**  enum boundary\_condition { Periodic = 0, Metallic, Magnetic, None };  **Field Function**  typedef std::complex<double> (\*field\_function)(const std::complex<double> \*fields, const vec &loc, void \*integrand\_data\_);  **chunkloop\_field\_components**  /\* A utility class for loop\_in\_chunks, for fetching values of field  components at grid points, accounting for the complications  of symmetry and yee-grid averaging. \*/  **fields**  int num\_chunks;  bool shared\_chunks;  fields\_chunk \*\*chunks;  src\_time \*sources;  flux\_vol \*fluxes;  symmetry S;  realnum \*\*comm\_blocks[NUM\_FIELD\_TYPES];  size\_t \*comm\_sizes[NUM\_FIELD\_TYPES][CONNECT\_COPY + 1];  double a, dt; // The resolution a and timestep dt=Courant/a  grid\_volume gv, user\_volume;  volume v;  double m;  double beta;  int t, phasein\_time, is\_real;  std::complex<double> k[5], eikna[5];  double coskna[5], sinkna[5];  boundary\_condition boundaries[2][5];  char \*outdir;  bool components\_allocated;  fields(structure \*, double m = 0, double beta = 0, bool zero\_fields\_near\_cylorigin = true);  void add\_point\_source(component c, const src\_time &src, const vec &, std::complex<double> amp = 1.0);  void add\_volume\_source(component c, const src\_time &src, const volume &where\_, std::complex<double> \*arr, size\_t dim1, size\_t dim2, size\_t dim3, std::complex<double> amp);  void add\_eigenmode\_source(component c, const src\_time &src, direction d, const volume &where, const volume &eig\_vol, int band\_num, const vec &kpoint, bool match\_frequency, int parity, double eig\_resolution, double eigensolver\_tol, std::complex<double> amp, std::complex<double> A(const vec &) = 0);  void initialize\_with\_nth\_tm(int n);  std::complex<double> integrate(int num\_fields, const component \*components, field\_function fun, void \*fun\_data\_, const volume &where, double \*maxabs = 0);  dft\_chunk \*add\_dft(component c, const volume &where, double freq\_min, double freq\_max, int Nfreq, bool include\_dV\_and\_interp\_weights = true,std::complex<double> stored\_weight = 1.0, dft\_chunk \*chunk\_next = 0, bool sqrt\_dV\_and\_interp\_weights = false,std::complex<double> extra\_weight = 1.0, bool use\_centered\_grid = true, int vc = 0);  double energy\_in\_box(const volume &);  double electric\_energy\_in\_box(const volume &);  double magnetic\_energy\_in\_box(const volume &);  double thermo\_energy\_in\_box(const volume &);  double total\_energy();  double field\_energy\_in\_box(const volume &);  **flux\_vol**  fields \*f;  direction d;  volume where;  double cur\_flux, cur\_flux\_half;  **harminv**  int do\_harminv(std::complex<double> \*data, int n, double dt, double fmin, double fmax, int maxbands, std::complex<double> \*amps, double \*freq\_re, double \*freq\_im, double \*errors = NULL, double spectral\_density = 1.1, double Q\_thresh = 50, double rel\_err\_thresh = 1e20, double err\_thresh = 0.01, double rel\_amp\_thresh = -1, double amp\_thresh = -1);  **Random**  // random number generation: random.cpp  void set\_random\_seed(unsigned long seed);  double uniform\_random(double a, double b); // uniform random in [a,b]  double gaussian\_random(double mean, double stddev); // normal random with given mean and stddev  int random\_int(int a, int b); // uniform random in [a,b)  **Bessel function**  (in initialize.cpp)  double BesselJ(int m, double kr);  **analytical Green's functions**  (in near2far.cpp); upon return,  // EH[0..5] are set to the Ex,Ey,Ez,Hx,Hy,Hz field components at x  // from a c0 source of amplitude f0 at x0.  void green2d(std::complex<double> \*EH, const vec &x, double freq, double eps, double mu,  const vec &x0, component c0, std::complex<double> f0);  void green3d(std::complex<double> \*EH, const vec &x, double freq, double eps, double mu,  const vec &x0, component c0, std::complex<double> f0); |
| MATERIAL\_DATA\_H | **material\_data**  struct medium\_struct {  vector3 epsilon\_diag;  cvector3 epsilon\_offdiag;  vector3 mu\_diag;  cvector3 mu\_offdiag;  susceptibility\_list E\_susceptibilities;  susceptibility\_list H\_susceptibilities;  vector3 E\_chi2\_diag;  vector3 E\_chi3\_diag;  vector3 H\_chi2\_diag;  vector3 H\_chi3\_diag;  vector3 D\_conductivity\_diag;  vector3 B\_conductivity\_diag;} |
| MEEP\_VEC\_H | const int NUM\_FIELD\_COMPONENTS = 20;  const int NUM\_FIELD\_TYPES = 8;  enum component {  Ex = 0,  Ey,  Er,  Ep,  Ez,  Hx,  Hy,  Hr,  Hp,  Hz,  Dx,  Dy,  Dr,  Dp,  Dz,  Bx,  By,  Br,  Bp,  Bz,  Dielectric,  Permeability,  NO\_COMPONENT  };  enum derived\_component {  Sx = 100,  Sy,  Sr,  Sp,  Sz,  EnergyDensity,  D\_EnergyDensity,  H\_EnergyDensity  };  **Vec**  double t[5];  ndim dim;  vec(double xx, double yy, double zz) {  init\_t();  dim = D3;  t[X] = xx;  t[Y] = yy;  t[Z] = zz;  };  vec operator+(const vec &a)  vec operator+=(const vec &a)  vec operator-(const vec &a)  vec operator-(void)  vec operator-=(const vec &a)  bool operator!=(const vec &a)  bool operator==(const vec &a)  vec round\_float(void)  vec operator\*(double s)  vec operator/(double s)  // I use & as a dot product.  double operator&(const vec &a)    double r() const { return t[R]; };  double x() const { return t[X]; };  double y() const { return t[Y]; };  double z() const { return t[Z]; };  **ivec**  **volume**  ndim dim;  vec min\_corner, max\_corner;  **grid\_volume**  The volume class declared in meep/vec.hpp represents a rectilinear region, parallel to the xyz axes, in "continuous space" — i.e. the corners can be at any points, not necessarily grid points. This is used, for example, whenever you want to specify the integral of some quantity (e.g., flux, energy) in a box-like region.  ndim dim;  double a, inva /\* = 1/a \*/;  grid\_volume(ndim d, double ta, int na, int nb, int nc);  ivec io; // integer origin ... always change via set\_origin etc.!  vec origin; // cache of operator[](io), for performance  int num[3];  ptrdiff\_t the\_stride[5];  size\_t the\_ntot;  **symmetry**  signed\_direction S[5];  std::complex<double> ph;  vec symmetry\_point;  ivec i\_symmetry\_point;  int g; // g is the multiplicity of the symmetry.  symmetry \*next;  friend symmetry r\_to\_minus\_r\_symmetry(double m);  **volume\_list** |
| Materials.py | **# default unit length is 1 um**  um\_scale = 1.0  **# conversion factor for eV to 1/um [=1/hc]**  eV\_um\_scale = um\_scale/1.23984193  **# copper (Cu)**  metal\_range = mp.FreqRange(min=um\_scale/12.398, max=um\_scale/.20664)  Cu\_plasma\_frq = 10.83\*eV\_um\_scale  Cu\_f0 = 0.575  Cu\_frq0 = 1e-10  Cu\_gam0 = 0.030\*eV\_um\_scale  Cu\_sig0 = Cu\_f0\*Cu\_plasma\_frq\*\*2/Cu\_frq0\*\*2  Cu\_f1 = 0.061  Cu\_frq1 = 0.291\*eV\_um\_scale # 4.261 um  Cu\_gam1 = 0.378\*eV\_um\_scale  Cu\_sig1 = Cu\_f1\*Cu\_plasma\_frq\*\*2/Cu\_frq1\*\*2  Cu\_f2 = 0.104  Cu\_frq2 = 2.957\*eV\_um\_scale # 0.419 um  Cu\_gam2 = 1.056\*eV\_um\_scale  Cu\_sig2 = Cu\_f2\*Cu\_plasma\_frq\*\*2/Cu\_frq2\*\*2  Cu\_f3 = 0.723  Cu\_frq3 = 5.300\*eV\_um\_scale # 0.234 um  Cu\_gam3 = 3.213\*eV\_um\_scale  Cu\_sig3 = Cu\_f3\*Cu\_plasma\_frq\*\*2/Cu\_frq3\*\*2  Cu\_f4 = 0.638  Cu\_frq4 = 11.18\*eV\_um\_scale # 0.111 um  Cu\_gam4 = 4.305\*eV\_um\_scale  Cu\_sig4 = Cu\_f4\*Cu\_plasma\_frq\*\*2/Cu\_frq4\*\*2  Cu\_susc = [mp.DrudeSusceptibility(frequency=Cu\_frq0, gamma=Cu\_gam0, sigma=Cu\_sig0),  mp.LorentzianSusceptibility(frequency=Cu\_frq1, gamma=Cu\_gam1, sigma=Cu\_sig1),  mp.LorentzianSusceptibility(frequency=Cu\_frq2, gamma=Cu\_gam2, sigma=Cu\_sig2),  mp.LorentzianSusceptibility(frequency=Cu\_frq3, gamma=Cu\_gam3, sigma=Cu\_sig3),  mp.LorentzianSusceptibility(frequency=Cu\_frq4, gamma=Cu\_gam4, sigma=Cu\_sig4)]  Cu = mp.Medium(epsilon=1.0, E\_susceptibilities=Cu\_susc, valid\_freq\_range=metal\_range)  #------------------------------------------------------------------  **# aluminum (Al)**  Al\_plasma\_frq = 14.98\*eV\_um\_scale  Al\_f0 = 0.523  Al\_frq0 = 1e-10  Al\_gam0 = 0.047\*eV\_um\_scale  Al\_sig0 = Al\_f0\*Al\_plasma\_frq\*\*2/Al\_frq0\*\*2  Al\_f1 = 0.227  Al\_frq1 = 0.162\*eV\_um\_scale # 7.654 um  Al\_gam1 = 0.333\*eV\_um\_scale  Al\_sig1 = Al\_f1\*Al\_plasma\_frq\*\*2/Al\_frq1\*\*2  Al\_f2 = 0.050  Al\_frq2 = 1.544\*eV\_um\_scale # 0.803 um  Al\_gam2 = 0.312\*eV\_um\_scale  Al\_sig2 = Al\_f2\*Al\_plasma\_frq\*\*2/Al\_frq2\*\*2  Al\_f3 = 0.166  Al\_frq3 = 1.808\*eV\_um\_scale # 0.686 um  Al\_gam3 = 1.351\*eV\_um\_scale  Al\_sig3 = Al\_f3\*Al\_plasma\_frq\*\*2/Al\_frq3\*\*2  Al\_f4 = 0.030  Al\_frq4 = 3.473\*eV\_um\_scale # 0.357 um  Al\_gam4 = 3.382\*eV\_um\_scale  Al\_sig4 = Al\_f4\*Al\_plasma\_frq\*\*2/Al\_frq4\*\*2  Al\_susc = [mp.DrudeSusceptibility(frequency=Al\_frq0, gamma=Al\_gam0, sigma=Al\_sig0),  mp.LorentzianSusceptibility(frequency=Al\_frq1, gamma=Al\_gam1, sigma=Al\_sig1),  mp.LorentzianSusceptibility(frequency=Al\_frq2, gamma=Al\_gam2, sigma=Al\_sig2),  mp.LorentzianSusceptibility(frequency=Al\_frq3, gamma=Al\_gam3, sigma=Al\_sig3),  mp.LorentzianSusceptibility(frequency=Al\_frq4, gamma=Al\_gam4, sigma=Al\_sig4)]  Al = mp.Medium(epsilon=1.0, E\_susceptibilities=Al\_susc, valid\_freq\_range=metal\_range)  **# nickel iron (NiFe) from Horiba Technical Note 09: Drude Dispersion Model**  # ref: http://www.horiba.com/fileadmin/uploads/Scientific/Downloads/OpticalSchool\_CN/TN/ellipsometer/Drude\_Dispersion\_Model.pdf  # wavelength range: 0.25 - 0.83 um  NiFe\_range = mp.FreqRange(min=um\_scale/0.83, max=um\_scale/0.25)  NiFe\_frq = 1/(0.0838297450980392\*um\_scale)  NiFe\_gam = 1/(0.259381156903766\*um\_scale)  NiFe\_sig = 1  NiFe\_susc = [mp.DrudeSusceptibility(frequency=NiFe\_frq, gamma=NiFe\_gam, sigma=NiFe\_sig)]  NiFe = mp.Medium(epsilon=1.0, E\_susceptibilities=NiFe\_susc, valid\_freq\_range=NiFe\_range) |
| C++ Tutorial | **Origin**  C++ interface by default puts the origin at the corner of the cell (0,0,0) to (LL,LL,LL)   1. include the definitions of the Meep routines. 2. initialize Meep, 3. and will generally then define a computational grid\_volume and the associated structure describing the geometry and materials, 4. initialize the fields, 5. add sources, 6. and then time-step   **grid\_volume**  Note the constructor for the grid\_volume class in 1d which takes as parameters the size of the cell and the resolution.  **sources**  The sources in Meep are excitations of the polarization vector in D=εE+P. The polarization can be any one of the six cartesian or cylindrical fields. There are a variety of sources including dipole and current sources, gaussian pulses and a continuous wave sources.  **H5topng**  To see what the unit cell of the dielectric function looks, output as an HDF5 file, we invoke the built in command:  f.output\_hdf5(Dielectric, v.surroundings())  This versatile command can be used to visualize all of the field components, field energy when called at a particular time step. The resulting output file will be eps-000000.00.h5 and to view it, we first need to convert it to the portable network graphics (PNG) format with the h5topng tool.  **time stepping of the fields**  and do so in a simple loop:  while (f.time() < f.last\_source\_time()) f.step();  **getfield**  f.get\_field(Ex,vol.center());  The get\_field function does exactly that, obtains the value of the field at a given position within the cell using linear interpolation where necessary  **Harminv**  obtain the frequencies of the modes which we do by invoking a special tool for harmonic inversion, harminv, to extract the real and imaginary frequencies:  int num = do\_harminv(p, ttot, f.dt, 0.8\*w\_midgap, 1.2\*w\_midgap, maxbands, amps, freq\_re, freq\_im);  The integer returned by harminv is the number of mode frequencies obtained from the input data p. The particular call to harminv included passing the arrays by value and telling harminv to look for frequencies within 20% of the mid gap frequency up to a maximum of 5 bands. At this point, the necessary information to compute the quality has been stored in the freq\_re and freq\_im arrays and we compute the quality factor using the formula, Q=−ωr/2ωi.  **Compiling**  Use the pkg-config program which is installed by default on most Linux systems:  g++ `pkg-config --cflags meep` foo.cpp -o foo `pkg-config --libs meep`  ./foo |
| Introduction | Meep implements the finite-difference time-domain (FDTD) method for computational electromagnetics. This is a widely used technique in which space is divided into a discrete grid and the fields are evolved in time using discrete time steps — as the grid and the time steps are made finer and finer, this becomes a closer and closer approximation for the true continuous equations, and one can simulate many practical problems essentially exactly.  **Maxwell’s Equations**  Meep simulates Maxwell's equations, which describe the interactions of electric (E) and magnetic (H) fields with one another and with matter and sources. In particular, the equations for the time evolution of the fields are:  dBdt=−∇×E−JB−σBB  B=μH  dDdt=∇×H−J−σDD  D=εE  where D is the displacement field, ε is the dielectric constant, J is the current density (of electric charge), and JB is the magnetic-charge current density. Magnetic currents are a convenient computational fiction in some situations. B is the magnetic flux density (often called the magnetic field), μ is the magnetic permeability, and H is the magnetic field. The σB and σD terms correspond to (frequency-independent) magnetic and electric conductivities, respectively. The divergence equations are implicitly:  ∇⋅B=−∫t∇⋅(JB(t′)+σBB)dt′  ∇⋅D=−∫t∇⋅(J(t′)+σDD)dt′≡ρ  Generally, ε depends not only on position but also on frequency (material dispersion) and on the field E itself (nonlinearity), and may include loss or gain.  **Units in Meep**  Meep uses dimensionless units where all these constants are unity. As a practical matter, almost everything you might want to compute (reflectance/transmittance spectra, frequencies, etcetera) is expressed as a ratio anyway, so the units end up cancelling.   1. Moreover, since c=1 in Meep units, a (or a/c) is the unit of time as well. 2. frequency f in Meep (corresponding to a time dependence e−i2πft) is specified in units of c/a (or equivalently ω is specified in units of 2πc/a), 3. which is equivalent to specifying f as 1/T: the inverse of the optical period T in units of a/c. 4. This, in turn, is equivalent to specifying f as a/λ where λ is the vacuum wavelength.   For example, we let a = 1 μm. Then, if we want to specify a source corresponding to λ = 1.55 μm, we specify the frequency f as 1/1.55 = 0.6452. If we want to run our simulation for 100 periods, we then run it for 155 time units (= 100/f).  **Boundary Conditions and Symmetries**  Three basic types of terminations are supported in Meep: Bloch-periodic boundaries, metallic walls, and PML absorbing layers. Also, one can exploit symmetries of a problem to further reduce the computational requirements. For example, if you know that your system has a mirror symmetry plane (both in the structure and in the current sources), then you can save a factor of two by only simulating half of the structure and obtaining the other half by mirror reflection.  **Finite-Difference Time-Domain Methods**  if the grid has some spatial resolution Δx, then our discrete time-step Δt is given by Δt=SΔx, where S is the Courant factor and must satisfy S<nmin/√#dimensions, where nmin is the minimum refractive index (usually 1), in order for the method to be stable (not diverge). In Meep, S=0.5 by default (which is sufficient for 1 to 3 dimensions), but can be changed by the user. This means that when you double the grid resolution, the number of time steps doubles as well (for the same simulation period). Thus, in three dimensions, if you double the resolution, then the amount of memory increases by 8 and the amount of computational time increases by (at least) 16.  **Yee Lattice**  in order to discretize the equations with second-order accuracy, FDTD methods store different field components at different grid locations. This discretization is known as a Yee lattice. As a consequence, Meep must interpolate the field components to a common point whenever you want to combine, compare, or output the field components (e.g. in computing energy density or flux). However, because it is a simple linear interpolation, while E and D may be discontinuous across dielectric boundaries, it means that the interpolated E and D fields may be less accurate than you might expect right around dielectric interfaces. If you ask for the Poynting flux through a certain rectangle, then Meep will linearly interpolate the field values from the grid onto that rectangle.  **Pervasive Interpolation**  the philosophy of the Meep interface is pervasive interpolation, so that if you change any input continuously then the response of the Meep simulation will change continuously as well, so that it will converge as rapidly and as smoothly as possible to the continuous solution as you increase the spatial resolution.  **Smoothing**  Meep's subpixel smoothing is specially designed in order to minimize the "staircasing" and other errors caused by sharp interfaces.  **Alternative Methods**  In cases where there are large differences in scale (e.g. with metals with a shallow skin depth), it may be better to use a method that allows a variable resolution in different spatial regions, such as a finite-element or boundary-element method.   1. Boundary-element methods are especially powerful when you have a large volume-to-surface ratio, such as for scattering calculations over small objects in a large (i.e., infinite-sized) volume. 2. Finite-element methods can also be used for time-evolving fields, but they suffer a serious disadvantage compared to finite-difference methods: finite-element methods, for stability, must typically use some form of implicit time-stepping, where they must invert a matrix (solve a linear system) at every time step. Finally, in systems that are composed of a small number of easily-analyzed pieces, such as a sequence of constant-cross-section waveguides, a collection of cylinders, or a multi-layer film, transfer-matrix/scattering-matrix methods may be especially attractive. These methods treat the individual simple elements in some analytic or semi-analytic fashion, enabling the entire structure to be simulated with great speed and accuracy. 3. A strength of time-domain methods is their ability to obtain the entire frequency spectrum of responses (or eigenfrequencies) in a single simulation, by Fourier-transforming the response to a short pulse or using more sophisticated signal-processing methods such as Harminv.   **Field Patterns and Green's Functions**  The field pattern from a given localized source at a particular frequency ω is a form of the Green's function of the system.  Gij(ω;x,x′)  which gives the ith component of (say) E at x from a point current source J at x′, such that J(x)=ej^⋅exp(−iωt)⋅δ(x−x′).  **Transmittance/Reflectance Spectra**  Computing the transmittance or scattering spectra from some finite structure, such as a resonant cavity, in response to some stimulus. One could, of course, compute the fields (and thus the transmitted flux) at each frequency ω separately. However, it is much more efficient to compute a broadband response via a single computation by Fourier-transforming the response to a short pulse.  Flux is the integral of the Poynting vector over the specified FluxRegion. It only integrates one component of the Poynting vector and the direction property specifies which component.  **Resonant Modes**  Another common task in FDTD is to compute resonant modes or eigenmodes of a given structure. |
| Field Functions | Meep provides several routines to integrate, analyze, and output arbitrary user-specified functions of the field components. |
| Tutorial/ Baiscs | **Transmittance/Reflectance Spectra**  In order to normalize the transmitted flux by the incident power to obtain the transmittance, we'll have to do two runs, one with and one without the bend (i.e., a straight waveguide). <https://github.com/NanoComp/meep/blob/master/python/examples/bend-flux.py>   1. computing the reflection spectra requires some care because we need to separate the incident and reflected fields. We do this by first saving the Fourier-transformed fields from the normalization run. 2. And then, before we start the second run, we load these fields, negated. The latter subtracts the Fourier-transformed incident fields from the Fourier transforms of the scattered fields. Logically, we might subtract these after the run, but it turns out to be more convenient to subtract the incident fields first and then accumulate the Fourier transform. 3. We need to keep running after the source has turned off because we must give the pulse time to propagate completely across the cell. Moreover, the time required is a bit tricky to predict when you have complex structures, because there might be resonant phenomena that allow the source to bounce around for a long time. Therefore, it is convenient to specify the run time in a different way: instead of using a fixed time, we require that |Ez|2 at the end of the waveguide must have decayed by a given amount (1/1000) from its peak value. This should be sufficient to ensure that the Fourier transforms have converged. 4. With the flux data, we are ready to compute and plot the reflectance and transmittance. The reflectance is the reflected flux divided by the incident flux. We also have to multiply by -1 because all fluxes in Meep are computed in the positive-coordinate direction by default, and we want the flux in the −x direction. 5. The transmittance is the transmitted flux divided by the incident flux. Finally, the scattered loss is simply 1−transmittance−reflectance. |
| Mode Decomposition | Consider a waveguide with propagation axis along the x direction and constant cross section in the transverse direction  . For a given angular frequency ω we can solve for the eigenmodes of the structure. Thus, arbitrary fields of the form E(r,t)=E(r)e−iωt and H(r,t)=H(r)e−iωt can be decomposed into a basis of these eigenmodes:    βn are the propagation wavevectors and α  ±  n  are the basis coefficients. Mode decomposition involves solving for these unknown quantities. In Meep, compute the Fourier-transformed fields E(r) and H(r) on a surface which is transverse to the waveguide and stored in a dft\_flux object. In MPB, compute the eigenmodes as well as the propagation wavevectors βn for the same cross-sectional structure.Compute the coefficients for any number of eigenmodes n=1,2,.... |
| Materials | **Material Dispersion** |
| Developer Information | **Functionality Organization**  Material dispersion polarization.cpp, update\_from\_e.cpp, and friends.  Vectors, volumes etc. meep/vec.hpp, vec.cpp  Geometric objects handled by libctl functions in libctl's geom.c, called from the Scheme front-end (not handled by Meep)  Fields: initialization, cleanup, chunking, stepping-plan, (dis)affiliation with sources, polarizabilities etc. fields.cpp  Structure: initialization, cleanup, chunking, material parameters, boundary conditions etc. structure.cpp |