Development Notes

v. 0.1

March 3, 2011

Contents

1 Introduction

2 Files Description

The source code of DISPHOTN consists of the following files:

- `disphotn.c`: the main program.
- `eigensolver.c`: the eigensolver code. This function uses a combination of the `ARPACK` and `PARDISO` libraries to find the generalized eigenvalues and eigenvectors of a couple of matrices aroung a target value.
- `file_io.c`: this file contains the functions that deal with files. The functions to import the mesh file and to write the results are here.
- `geometry_manipulation.c`: this file contains geometry-related functions, used to identify edges, boundaries, periodic conditions etc.
- `matrix_building.c`: this file contains the function that computes the elemental matrices for a mesh element.
- `matrix_manipulation.c`: this file contains the functions used to perform matrix operations and manipulation.
- `structs.c`: this file contains the data structures used in the program.
- `utility.c`: this file contains utility functions used through the program.

3 Dependencies

DISPHOTN depends directly on the functions contained in the following libraries:

- `ARPACK` (ARnoldi PACKage, `http://www.caam.rice.edu/software/ARPACK/`): an open-source eigensolving library.
- `PARDISO` (`http://www.pardiso-project.org`) a proprietary sparse linear system solver.

These packages, in turn, use the following libraries:

- `BLAS` (Basic Linear Algebra Subprograms): a collection of linear algebra routines.
- `LAPACK` (Linear Algebra PACKage): routines for the solution of linear algebra problems.

Since all these packages are written in `FORTRAN`, the following libraries are also required:

- `libgfortran`,
- `libgomp`.

4 Anatomy of a Simulation

In this section the program flow will be described in detail. The general structure of the 2D and 3D programs is the same, the minimal differences will be pointed out when needed. In general, the functions that differ between the two programs have a `_3d` appended in the name for the 3D version, for example, the function `identify_edges` in the 2D program is called `identify_edges_3D` in the 3D program.

4.1 Data Structures

Before discussing the simulation process itself, a word must be spent on the data structures used in the program.

- `bool`: this data structure is actually a char, but it is a logical construct to express that the variable may only assume the value of false (0) or true (-1).
- `byte`: this represent a short integer, that should be able to assume the values of 1, 0, and -1. This structure is also a char.
- `doublecomplex`: this struct represents a complex number in double precision and is made up by two doubles, one for the real part and one for the imaginary part. This data structure is shared with the `FORTRAN` routines and should not be altered.
- `strlist`: a list element composed of a pointer to itself and a pointer to a char. Used to create the string lists from the configuration file.
- `point2d` or `point3d`: a structure representing a point in the 2D or 3D space. Has two or three double values representing the Cartesian components.
- `field2d` or `field3d`: represents a vector field in 2D or 3D space. It is made up by two or three `doublecomplex` values representing the Cartesian components of the vector field.
- `edge`: represents a mesh edge. Has two integer fields that store the two node indices and a double field containing the length of the edge.
- `triangle` (2D only): represents a 2D mesh element. It contains 3 integer fields with the indices of the nodes; an integer containing the domain number; 3 integers containing the edge indices and 3 bytes containing the orientation of the edges.
- `tetra` (3D only): represents a 3D mesh element. It contains 4 integer fields with the indices of the nodes; an integer containing the domain number; 6 integers containing the edge indices and 6 bytes containing the orientation of the edges.

4.2 Configuration File

The first step is the reading of the configuration file name. If not specified in the command line, the program will ask it from the user. The function `read_configuration` is then called. This function will open the configuration file, strip it of all the comments (beginning with the character #) and blank lines, and return two lists of strings. The first list, pointed by the `variables` pointer, contains the first word of each line (that should be the name of the parameter/flag/sequence etc.), while the second, pointed by `values`, contain the rest of each line. The program then scrolls through the list, comparing the strings in `variables` to valid keywords, and setting the parameters, sequences or flags according to the values in `values`. Once all the elements of the lists have been processed, the program checks the validity of the parameters inserted and the presence of the required parameters.

4.3 Interpolation Grid

If a grid file is specified in `gridfile`, the program imports the interpolation points by calling the `read_grid_file_2d` (or `_3d`) function. Otherwise, a regular grid with the same shape of the

unit cell is computed from the primitive vectors `a1`, `a2` (and `a3`), using the values in `x-`, `y-` and `zgrid`. In this case, the point in the position (i, j, k) is computed as:

$$\boldsymbol{p}\_ijk = \frac{i\boldsymbol{a}\_1}{(\texttt{xgrid}-1)} + \frac{j\boldsymbol{a}\_2}{(\texttt{ygrid}-1)} + \frac{k\boldsymbol{a}\_3}{(\texttt{zgrid}-1)}, \qquad i = 0, 1, \ldots \texttt{xgrid}-1, \text{etc.} \quad (1)$$

The number of grid elements is stored in the integer variable `gridn`, while the coordinates are stored in the `grid` array.

4.4 Mesh Reading

The next step is the mesh file reading. The `read_mesh_file` (or `read_mesh_file_3d`) function is called and the information in the mesh file is stored in memory. The following data is read from the mesh file, the rest is ignored:

- Nodes data: the number of nodes and the coordinates of each node.
- Triangle data (2D problems): the number of triangles, the nodes constituting each triangle and the domain it belongs.
- Tetrahedron data (3D problems): number, nodes and domains, as above.

The relevant information is returned in the `points` and `tries` (or `tetras`) arrays.

4.5 Edge Identification

Since the mesh file does not report information on the edges (it is designed for a node-based FE approach), the edges must be identified and associated with each triangle. This is done by the `identify_edges` function. This function returns the array `edges` of the unique edges in the geometry and updates the `triangles` (or `tetras`) array by saving, for each element, the number of the edges it contains.

4.6 Geometrical Computations

In this section, the program computes basic geometrical information used in the program, for example, the area of the triangles (or the volume of the tetrahedra), the edge length and their directions.

4.7 Boundary Identification

This part identifies the edges on the boundary (i.e. those on which the boundary conditions will be applied). To do so, the program cycles over the four (2D) or six (3D) boundaries, and for each boundary the following procedure is used:

- For 2D problems: for each edge, a vector `par2` is subtracted from the coordinates of each point. If the resulting two points are collinear with a vector `par1`, the edge is added to the current boundary.
- For 3D problems: for each edge, a vector `par3` is subtracted from the coordinates of each point. If the resulting two points are in the plane spanned by the vectors `par1` and `par2`, the edge is added to the current boundary.

Different values of the comparison vectors `par1`, `par2` and `par3` give the edges on different boundaries. For example, in a 3D problem, for `par1 = a1`, `par2 = a3`, `par3 = a2`, the algorithm retrieves the edge on the boundary opposite to the one spanned by `a1` and `a2`. The edges belonging to each boundary are stored in the two-dimensional array `boundary`, in which the i-th row contains the number of the edges on the i-th boundary.

Note that if the wrong primitive vectors are specified, one or more boundaries have zero edges on them; moreover, if periodic conditions are required, the opposite boundaries must have the same number of edges. The program checks that these two conditions are fulfilled before continuing.

During the identification, the edges belonging to a PEC boundary (that is, a boundary on which the Perfect Electric Conductor boundary conditions must be imposed) are identified as well; the array `on_PEC` is created in which the i-th element is set to `TRUE` if the i-th edge is on a PEC boundary

4.8 Periodic Conditions

If Periodic conditions are required, the program identifies the corresponding edges on opposite boundaries. The function `identify_periodic_boundaries` checks that each edge on a periodic boundary has the corresponding edge on the opposite boundary. The function reorders the edge numbers in the `boundary` array such that the corresponding edges are in the same place in the array. For example, if the boundary 0 is made up by the edges 1, 3, 7 and the boundary 2 by the edges 2, 5, 9, before the call to `identify_periodic_boundaries` the bidimensional array `boundary` may look like this:

$$\begin{array}{l} \text{boundary}[0] = \{1,\ 3,\ 7\} \\ \text{boundary}[1] = \ldots \\ \text{boundary}[2] = \{2,\ 5,\ 9\} \\ \ldots \end{array}$$

After the call to the function, if the corrispondence between boundaries is $1 \to 5$, $3 \to 2$, $7 \to 9$, the array will be sorted in the following way.:

$$\begin{array}{l} \text{boundary}[0] = \{1,\ 3,\ 7\} \\ \text{boundary}[1] = \ldots \\ \text{boundary}[2] = \{5,\ 2,\ 9\} \\ \ldots \end{array}$$

The function `identify_periodic_boundaries` returns also another array, called `boundary_orientations_x` containing the relative orientations of the edges from the boundary `x` to the boundary `y`

4.9 Interface Identification (2D only)

To apply the penalty method for the interface conditions, the edges belonging to more than one domain must be identified. The array `interface_info` will contain the domain number of the two triangles bounding each edge. The edges belonging to an interface will be those who have two different domain numbers.

4.10 Metallic Edges Identification (3D only)

Due to performance reasons (mainly because in principle any number of tetrahedra may share the same edge), the interface analysis described in the previous section is not carried on for 3D problems. Instead, the program will record in the `metal_info` array only the metallic domains each edge belongs to. The `metal_info` is an array of length (number of edges) $\times$ (number of metallic domains). If the i-th edge belongs to the j-th metallic domain, then the value of the element $j\times$ (number of edges) $+i$ of `metal_info` is set to `TRUE`.

4.11 Matrix Construction

At this point, all the information required has been retrieved and the FE matrices can be built. In this section, the space for the matrices is allocated and the values are inserted in a naive manner. The generic matrix X (X may be A or B) is described by the three arrays `ix`, `jx` and `x`. Each triplet of values at the position i of each array (namely, `ix[i]`, `jx[i]` and `x[i]`) form a *matrix entry*, composed by the row and column index and the value of the element.

The construction of the matrix follow this procedure: for each element of the mesh, the elemental matrix (3×3 for triangles and 6×6 for tetrahedra) is computed and its entries are added to the `ix`, `jx` and `x` arrays, taking into account the edge orientations for each edge. At the end of this process, the arrays would have length (number of elements)×(number of edges per element)^2. However, for each metallic domain, new degrees of freedom corresponding to the $\boldsymbol{P}$ vector are added, thus increasing the length of the arrays.

The elemental matrices are computed by the `matrix_construction` function, and the matrix entries are added using the `add_to_matrix` function, that takes care of positioning each entry on the right row and column, and applying coefficients due to the material parameters.

For 2D problems only, additional entries are created from the imposition of the interface penalty method. For 3D problems, after the matrix construction process is complete, the OP matrix is extracted from the F matrix.

After this step, most (if not all) the matrix entries in the arrays are duplicated, i.e. there are more than one entry for element. The entries pointing to the same matrix element must be summed together. To do this, first the three arrays are sorted using the `sort_matrix_entries` function (an implementation of the Quicksort algorithm) following the row number (`ix`) first and the column number (`jx`) second. This semplifies the next step, performed by the `assembly_matrix` function, that is summing all the duplicated matrix entries. Due to the sorting, the duplicated entries are now in consecutive positions in the arrays, and the function simply sums them. At the end of the function, the arrays are resized (reallocated) to their actual size (much shorter, since now all the entries are unique).

4.12 PEC Boundary Conditions

The two kinds of boundary conditions have different properties, in particular, the Periodic Boundary conditions depend on the value of the Bloch $\boldsymbol{k}$ vector, and therefore must be applied later (during the band structure computation). The PEC boundary conditions, instead, can be applied directly at this point of the program. This kind of condition simply correspond to setting to zero the value of the field on the edges belonging to PEC boundaries. The `set_PEC_conditions` function, therefore, sets to zero the rows and colums of the matrices A and B corresponding to edges on a PEC boundary (using the information stored in the `on_PEC` array). Note that the matriex entries are not removed from the matrices, but they are simply set to zero.

4.13 Band Computation (Cycling over $\boldsymbol{k}$ values)

At this point, the second kind of boundary conditions must be imposed. In this case, anyway, the conditions depend on the value of the vector $\boldsymbol{k}$. The following part of the program will then be repeated for each k-point specified in the configuration file.

Since the matrices are heavily altered during the solution process, a copy need to be made. The six arrays containing the information on the A and B matrices are copied to six new arrays (with a `k` appended to the name), that will be manipulated during the cycle.

4.14 Periodic Boundary Conditions

The periodic boundary conditions, depending on the current value of $\boldsymbol{k}$, are now applied. The procedure is quite simple, and uses the information in the `boundary` array. All the entries belonging to a row or a column on a periodic boundary are multiplied by a Bloch phase factor $\exp(i\boldsymbol{k}\cdot\boldsymbol{r})$ and moved on the row or column belonging to the corresponding edge. This reflects the fact that this set of degrees of freedom are related to another set of degrees of freedom via multiplication of a phase factor. After this step, the functions `sort_matrix_entries` and `assembly_matrix` are called again to ensure uniqueness of the entries.

4.15 Matrix Resizing

At this point, the matrices are ready, but there are empty rows and columns due to the imposition of the boundary condition and due to the fact that not all the edges belong to the metallic domains (and therefore the $\boldsymbol{P}$ vector is zero). The functions `purge_matrix` take care of resizing the matrices removing all the empty rows and columns, and producing an array `keep_indices` containing the indices of the nonempty rows and columns in the original matrices. The size of this array, contained in `nKeep`, will be the final size of the problem.

4.16 Eigensolving

Before starting the actual eigensolving process, one last task must be performed. `PARDISO` and `ARPACK` work on matrices in Compressed Row Storage, so the function `compress_rows` must be called on the matrices before passing them to the eigensolver.

At this point the `eigensolver` function is called, passing the matrices and all the relevant information, as well as the pointers to the arrays that will contain the results of the simulation.

4.17 Postprocessing

After the required eigenvalues and eigenvectors are obtained, the program performs postprocessing of the results. The missing degrees of freedom are reconstructed either by setting them to zero (for the PEC boundaries) or computing them using Bloch's theorem. Then, the field is interpolated on the interpolation grid specified using the `interpolate` function. For 2D problems, the z component of the electric field is computed for each triangle, using the curl theorem.

The computation for the current k-point is now completed and the program will now continue with the next k-value.

4.18 Outputting Results

After all the k-points have been computed, the program outputs the results obtained (eigenfrequencies, modes, and, for 2D problems, computed magnetic field) writing the data on the files specified in the configuration.

5 Discussion, Problems and Future Work

In the present section the problems currently affecting the program will be analyzed and discussed, and some suggestions will be made on possible improvements.

5.1 Interface Spurious Modes

The interface of metallic media introduce a serious problem in the simulation results. From the theory of surface plasmons, there are an infinite number of eigenmodes corresponding to electromagnetic field confined at the metallic surface, and their eigenfrequencies should converge to a value that will be called ω_s with increasing mode order. The discretized problem should show a similar behavior, i.e. a high number of modes concentrated around ω_s, that increase with refining the spatial discretization. Unfortunately, spurious modes localized at the interface appear when discretizing the problem. I found this issue first with a finite difference approach, using the Yee grid. I moved to a Finite Elements method, and decided to use edge-based elements, because of the encouraging results obtained with the "penalty method", described later. However, I could not find a better solution that does not radically alters the eigenfrequencies of the modes and at the same time eliminates the spurious modes at the interfaces. Observing the field plot for the spurious modes, it seems that they are caused by the failure of the imposition of correct interface condition between the fields, mainly the normal continuity of the $\boldsymbol{D}$ vector. The edge elements naturally impose the tangential continuity of the electric field, however, they have been used for the polarization field as well, where tangential continuity is not required (actually, imposing it is an error). Directly imposing the interface conditions (i.e. expressing some of the fields at the interface in terms of the others does not help in the removal of the spurious modes: the non-continuity is simply moved from the interface edges to other edges. Using higher-order edge elements seem not to be a solution. The problem of the spurious interface modes is retained even with first-order edge elements (9 DOF per triangle).

In my opinion, a solution to this problem must pass through a change of element type, at least for the polarization field. A mixed method, with edge-elements for $\boldsymbol{E}$ and node-elements for $\boldsymbol{P}$, may be the first attempt. The use of nodal elements for P can allow for an quite easy imposition of the interface conditions. However, the edge-elements have the well-know property of not introducing spurious modes inside the domains; this may be affected chosing another element type.

5.2 Penalty Method

The current version of the program uses a sort of "trick" to impose the correct interface conditions. Unfortunately, this seem to critically affect the eigenfrequencies of the higher order modes, increasing their value, as well as reducing the convergence rate. A brief explanation of the method used is the following. According to the theory, Eq. (2) is valid in the elements (triangles, tetrahedra) belonging to metallic regions,

$$\omega^2 \begin{bmatrix} \boldsymbol{E} \\ \boldsymbol{P} \end{bmatrix} = \begin{bmatrix} \frac{1}{\epsilon_0}\nabla \times \frac{1}{\mu_0}\nabla \times \bullet + \omega_p^2 \bullet & -\frac{\omega_0^2}{\epsilon_0}\bullet \\ -\omega_p^2 \epsilon_0 \bullet & \omega_0^2 \bullet \end{bmatrix} \begin{bmatrix} \boldsymbol{E} \\ \boldsymbol{P} \end{bmatrix}, \tag{2}$$

while Eq. (3) holds in the free-space or dielectric regions.

$$\nabla \times \frac{1}{\mu_0}\nabla \times \boldsymbol{E} = \omega^2 \epsilon_0 \epsilon \boldsymbol{E}. \tag{3}$$

The weak formulation of the problem requires an integration over the element, and the discretization of the problem is carried on expressing the field inside the element as a discrete sum of basis functions,

$$\boldsymbol{E} = \sum_i \boldsymbol{N}_i e_i \tag{4}$$

where e_i are the degrees of freedom of the finite elements method. The elemental matrices for the element are obtained from those integrals. The "penalty method" consists in identifying

the edges of a dielectric element that are shared with a metallic element, and computing the diagonal element of the elemental matrix corresponding to that edge *as if it were in the metal*. This has no physical meaning, and this is probably the reason why the eigenfrequencies are altered. Optionally, a coefficient may be introduced to reduce the impact of this method (this is how it works in the 2D problem).

5.3 Eigenvalue Computation

The program has been rewritten in C from the original `MATLAB` version to take advantage of the `PARDISO` library. The eigensolving algorithm used is `ARPACK`, an iterative method. To find eigenvalues around a target frequency (as is our case, since usually we are interested to the surface plasmons), the shift-and-invert approach is used. This requires a way to compute the matrix-vector product $y = (A - \sigma B)^{-1}x$; of course, since the matrices A and B can have potentially tens of thousand rows and columns, the direct invertion of the matrix is not viable, and therefore an alternative way is to solve $(A - \sigma B)y = x$. `MATLAB` (in the `eigs` function) uses a QR factorization of the matrix; while this is a good and fast method in many cases, if the matrices grow beyond a certain size, the memory requirement becomes prohibitive. I decided to use the iterative linear system solver `PARDISO` instead. This dramatically cuts the memory usage, but the computation time is increased. The eigenvalues returned by `ARPACK` are, of course, the same with both the methods.