Term Project

Combined-Method Calculations in Electromagnetics

Yulia Smirnova

08.08.2012

1. Problem Description

The standard numerical approaches used in electromagnetics – for example Fourier Modal Method (FMM [1]) or Finite Difference Method (FDM) – do not work well for complicated geometries or massive 3D objects. Using FDM one gets a pronounced staircase effect, besides FDM does not work well on highly resonating structures. And FMM works best on waveguides and is just not suited for complex models. Thus one is "forced" to use Finite Elements Method (FEM) to get more reasonable solutions. This method however is very computationally expensive in comparison with FDM and FMM.

Since using FEM is not always optimal, another approach is being used: when geometry can be represented as a number of blocks of different type, one can use different solvers for each part and at the end simply assemble the solutions together via scattering matrix – *S-matrix*.

The example of such a structure can be seen on the figure 1.

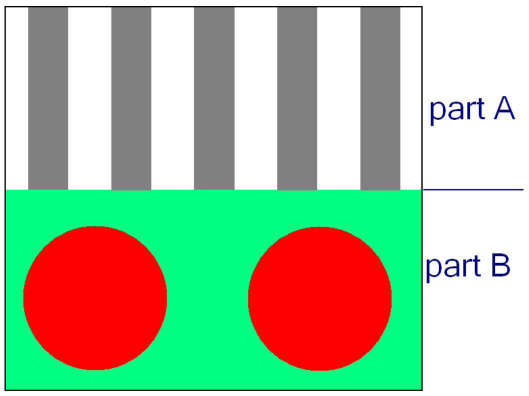

Figure 1: Example of a model for a "combined methods" approach

FMM is faster for layers homogeneous in one direction, i.e. for the part A on the picture. FEM however is more suitable for complex geometries, such as part B. Using FEM in part A demands unnecessarily fine mesh. Thus for this geometry a combination of two methods – FMM and FEM – works and could make the solution faster and more precise.

The software that works with S-matrices from different solvers has already been implemented by A.Dorodnyy IFH ETH Zurich. The goal of this work was to study a free software package *FreeFem++* and using this package to create an S-matrix for combining two methods – FMM and FEM.

2. FreeFem++: General Desciption

FreeFem++ is a programming language and a software focused in solving partial differential equations using FEM [2].

The procedure of creating a model in FreeFem++ is the following:

1. create geometry;
2. create mesh;
3. choose basis functions type;
4. write the weak formulation of the equation in the integral form;
5. solve.

Basically only weak formulation is left for the user and demands some kind of preparation. The rest is done inside the code.

The weak formulation is still written in the integral form, with no matrices involved. For example for the basic Helmholz equation

$$\Delta E + \epsilon\omega^2 E = 0 \tag{1}$$

the weak form looks like this:

$$\int_D (\bigtriangledown E \cdot \bigtriangledown v - \epsilon\omega^2 E v) dS - \int_{\partial D} v \bigtriangledown E \cdot dl = 0,\ \forall v \in S, \tag{2}$$

where S – is the basis we choose.

And in the FreeFem++ code this line looks like this:

```
problem HELMHOLZ(E,v) = int2d(D)( dx(E) * dx(v) + dy(E) * dy(v))
                      - int2d(D)( omega^2 *  eps(x,y) * E * v)
                      - int1d(D, border1)(v * dx(E) * n.x)
                      - int1d(D, border1)(v * dy(E) * n.y)
                      + on(border2, E=0)
```

Since in FreeFem++ we cannot modify basis functions to impose Dirichlet boundary conditions, they have to be included into the weak form. This is seen in the last line of the example above.

FreeFem++ offers several bases of ordinary and vector functions. In this study we used ordinary polynomial functions of the 2^{nd} and 3^{rd} order for E-polarization and vector polynomial functions for H-polarization.

The program really eases the usage of the FEM and is very well suited for simple 2D problems, but is quite limited once one wants to solve some complicated problems or go to 3D space.

Besides there is almost no control over the mesh generation: in order to make some special mesh changes one has to add geometrical curves and it is not always feasible and definitely not always easy, since adding one curve may change the other curves, mesh parameters and even domains of the weak form.

Moreover the build-in precision of the program turned out to be quite small, for instance π is only given up to 5 digits, this might not only give small precision of the output, but also lead to erroneous results.

Lastly the programm has a very poor graphic interface. Control over the output images is limited, though one can always use gnuplot functions to plot something special.

3. FMM: General Description

The method is based on the Fourier expansion of the Maxwell equations in xy-plane, assuming homogeneity in z-direction. If the space is inhomogeneous then we have to split the space into layers [1].

Thus in classic FMM, the structure is divided into several layers, we then solve Maxwell's equations in each layer independently, expanding all the variables in Fourier series, and combine the layers using an S-matrix formalism.

Where the S-matrix – scattering matrix – relates the outgoing waves b_1, b_2 to the incoming waves a_1, a_2 that are incident on the two-port:

$$S = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix}, \quad \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \tag{3}$$

The matrix elements S_{11}, S_{12}, S_{21}, S_{22} are referred to as the scattering parameters or the S-parameters. The parameters S_{11}, S_{22} have the meaning of reflection coefficients, and S_{21}, S_{12}, the meaning of transmission coefficients.

The resulting S-matrix provides the opportunity to calculate a variety of interesting properties such as transmission, reflection, absorption, diffraction, and near-field distribution as well as the eigenmodes of the structure. The S-matrix method is based on the eigenmodes of each layer that can propagate with an effective wave number K in the direction perpendicular to the layers (i.e. the z-direction).

The results of FMM depend highly on the Fourier expansion accuracy.

4. Model Description

For the test model we have chosen a three-layered structure depicted on the fig.2, which schematically represents a solar-cell structure. The top layer is TCO (transparent conducting oxide), followed by a layer of silicon. Technically there should be two silicon layers forming a pn-junction, but from the point of view of optical properties both n-dopped and p-dopped layers are the same, so we model them like one. At the bottom of the original model there is a layer of back reflector. Such structure can be and was calculated using FMM only.

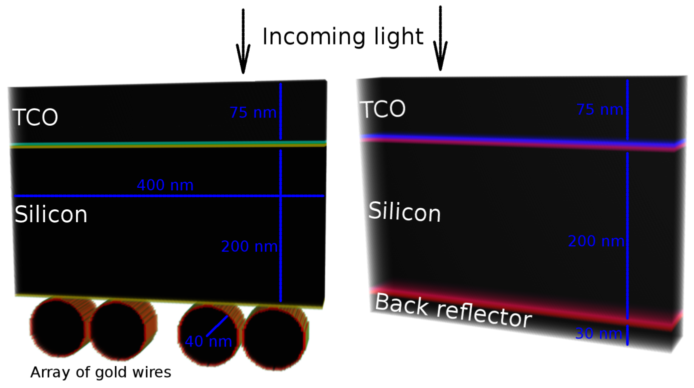

Figure 2: Test model

However if we substitute the last layer by an array of golden-wires the model becomes too complicated for FMM. The FMM results for the modified model show, that this method is good enough for zero-modes, but its precision drops off significantly for non-zero modes, demanding calculation of too many modes, especially in 3D.

The reason for substitution of the back-reflector layer by a wire-array is that such arrays significantly enhance the absorbtion in the silicon layer (fig.3).

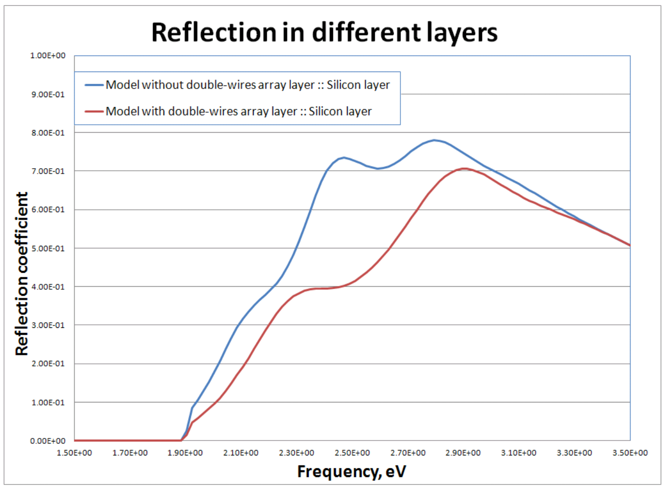

Figure 3: Effect of the wire-array layer (FMM results)

Our model is 2D and using FEM in this case is not much faster than using FMM only, but already in such a simple model we can see how much better the precision of the solution is, when we use FEM compared to FMM results.

We split the model into 3 parts – a part for each layer – and model the last layer – the wire-array – using FEM.

Thus our FreeFem++ model consists of a golden double-nanowires array surrounded by air. Diameter of the wires is 40 nm, the gap between them varies and equals 1,2,5 and 10nm. The structure is periodic in horizontal direction with the period 200nm. The dimensions can bee seen on the fig.4.

5. FreeFem++ model description

The model is 2D. We only model one period length in horizontal direction,

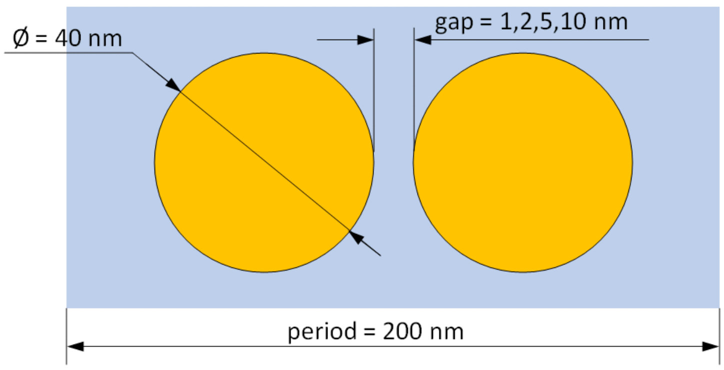

Figure 4: Test model for FreeFem++

setting periodic BC on the lateral sides.

In order to simulate unlimited space in vertical direction and not to have reflected waves from horizontal boundaries we add PML layers on the top and on the bottom of the structure. The thickness of the PML layer was chosen to balance the reflection and minimize the numerical efforts [3].

We are free to choose the amount of air above and below cylinders. And follow the general ideas: it should be large enough to prevent evanescent waves from reaching PML layers, and it should be small enough to decrease numerical efforts.

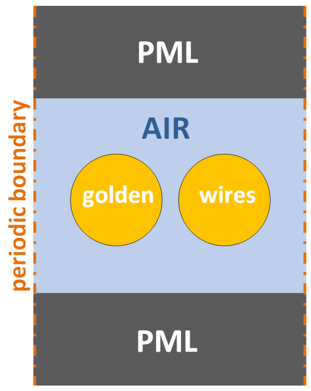

Figure 5: FreeFem++ model

The most interesting zone of the problem is the zone between two cylinders – it is a problematic zone for H-polarization case due to plasmonic resonance. Thus we refined the mesh there. An example of the coarse mesh of the model can be seen on the fig. 6.

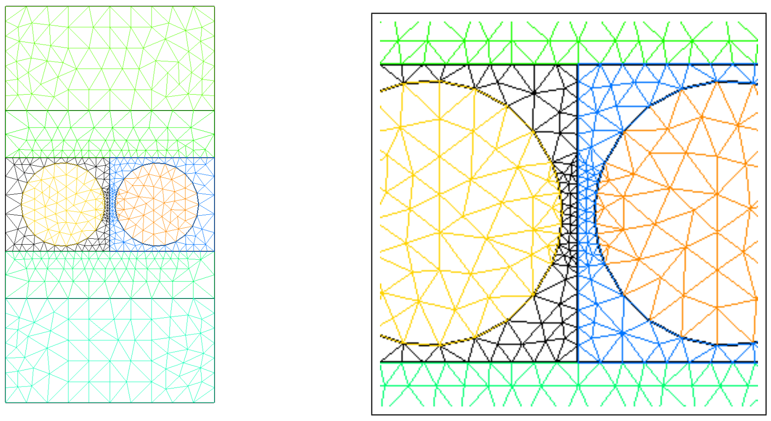
Figure 6: Example of the mesh

Additionally to extract data for the Fourier Transform (FT) (necessary for S-matrix construction) two straight output lines were added in the geometry of the model (see fig.7). Ideally they have to be placed on the top of the cylinders, but in FreeFem++ it is not possible. Hence we had to place them one element apart and then make a correction of modes' amplitudes and phases during FT calculation.

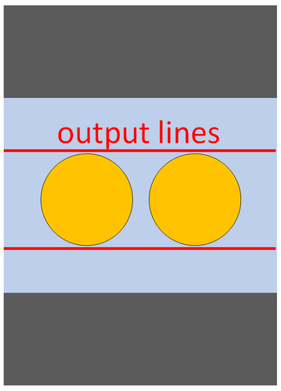

Figure 7: Output lines

The source of excitation in the model are induced waves. And we solve our equations to determine the scattered field.

The equations that we solve are the ordinary Maxwell equations in frequency domain, i.e. :

$$\int_D (\mu \bigtriangledown E \cdot \bigtriangledown v - \epsilon\omega^2 Ev + (1-\epsilon)\omega^2 k_y E_{inc} v)dS = 0, \ \forall v \in S, \qquad (4)$$

$$\int_D (\mu\ curl_{2D}\overline{E}\ curl_{2D}\overline{v} - \epsilon\omega^2\overline{E}\cdot\overline{v} + (1-\epsilon)\omega^2 E_{inc}\overline{k}\cdot\overline{v})dS = 0,\ \forall v \in S, \quad (5)$$

$$E_{total} = E + E_{inc}; \quad (6)$$

where

$\overline{k} = (k_x, k_y)$ – wave vector,

E – electric field,

E_{inc} – incident electric field,

ω – wave frequency,

$\mu = \mu(x, y)$ – material permeability,

$\epsilon = \epsilon(x, y)$ – material permittivity,

v and $\overline{v}$ - scalar and vector test functions.

Equation (4) is for the case of E-polarization, equation (5) is for the case of H-polarization and (6) is just a calculation of the desired scattering field.

The boundary conditions are the following:

- periodic on the lateral sides;
- $E = 0$ on the top and bottom boundaries.

The FreeFem++ script creates the model and calculates all the necessary cases for creating S-matrix. As input variables it takes the frequency value – ω – and the desired number of modes to calculate – N. Altogether it calculates $4N + 2$ cases: $k_x = 0, k_y = \pm\omega$ and $k_x = ik, k_y = \pm\sqrt{\omega^2 - k_x^2}$ for $i = 1, ..., N$, – and at the end it writes an output file with data from the output lines.

The FreeFem++ code-file is given in **Appendix A**.

6. Calculations

For each gap size we make calculations in FreeFem++ for TE and TM modes separately.

Each time frequency varies between 1 and 6 eV in 100 steps.

We want to obtain an S-matrix for each case, thus after each calculation we expand the resulting E-field in Fourier series. For this we collect E-values from the nodes on two output lines (see fig.7). These nodes were equidistantly placed to allow us perform Fourier transform (using C++ *fftw3.h* library). Sorting the result so that the modes would go in the desired order we finally fill in the entries of the matrix S from (3) in the following way:

- S_{11} – corresponds to the reflection from the top up;
- S_{12} – corresponds to the transmission from the bottom up;
- S_{21} – corresponds to the transmission from the top down;
- S_{22} – corresponds to the reflection from the bottom down.

The material matrix F is used to obtain the amplitudes from the electric field.

A separate C++ script does all this work, i.e. takes the output file of FreeFem++, performs Fourier transform for each calculated case and stores the result row by row in the form of the S-matrix. The C++ code-file is given in **Appendix B**.

7. Results

7.1. Gauging model

Before making a joint FEM–FMM calculation the results of the FreeFem++ were compared with the reference – BASE – solutions, which were obtained using CST Studio and MMP (Multiple Multipole Program) developed by prof. Christian Hafner IFH ETH Zurich. Both these software packages are very reliable and can be trusted to be a good reference. (The solution precision is valid at least up to the 5th digit).

We have the following models:

- $gap = 1, 2, 5, 10$nm;
- E and H polarizations;
- reflection and transmittion of the waves;
- mesh size: setting *scale* parameter 1,2,3,4;
- different k_x and k_y.

Thus we had quite a lot of cases to make a reliable comparison. The results match quite well. Below we present a couple of plots, that illustrate this (see fig.8 and 9).

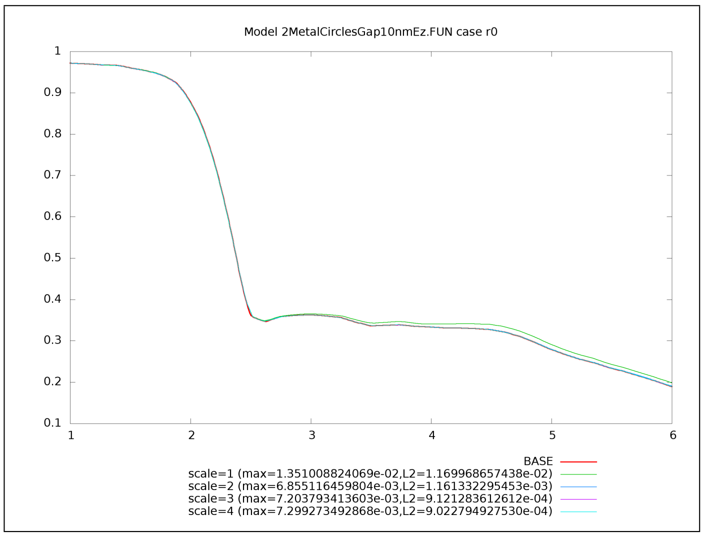

Figure 8: E-polarization, gap $=$ 10 nm, $k_x = 0,\ k_y = \omega$, reflection

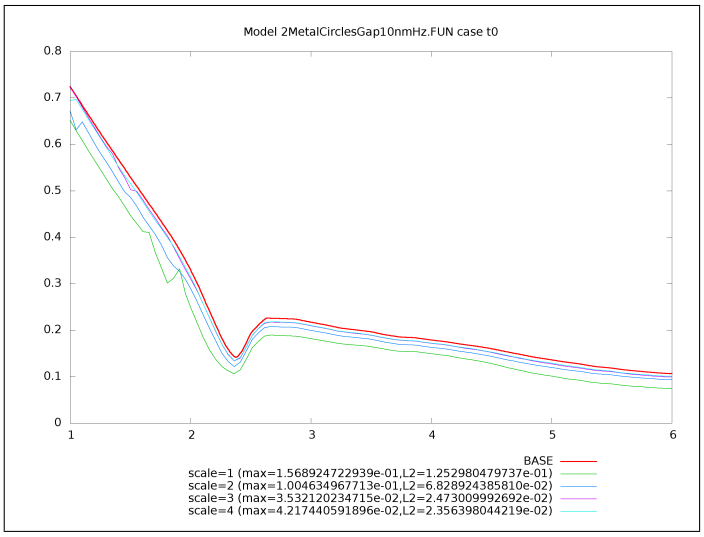

Figure 9: H-polarization, gap $=$ 10 nm, $k_x = 0,\ k_y = \omega$, transmission

To generalize our results we calculated L_{inf} and L_2 errors for our solutions and present the latter on the plots below (fig.10 and 11).

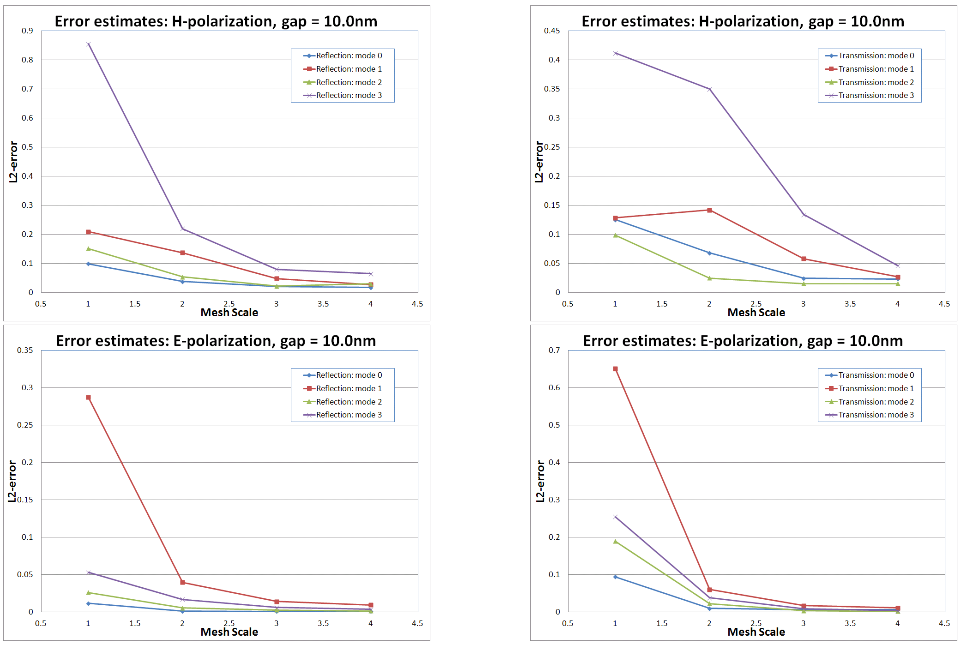

Figure 10: L2 Error, model with gap = 10.0 nm

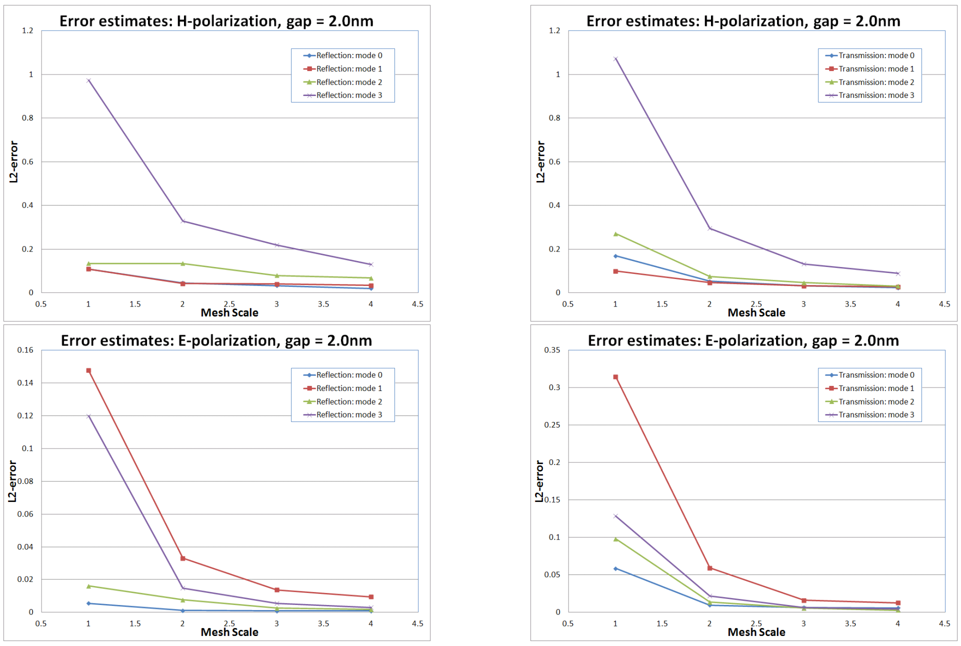

Figure 11: L2 Error, model with gap = 2.0 nm

We can see that the results in the case of E polarization match the BASE data better, whereas H polarization converges worse. But this is due to plasmonic resonance and was to be expected.

Such results and such high errors may seem unsatisfactory. But in fact they are not, since the alternative method – i.e. FMM – gives high oscillations (see fig.12).

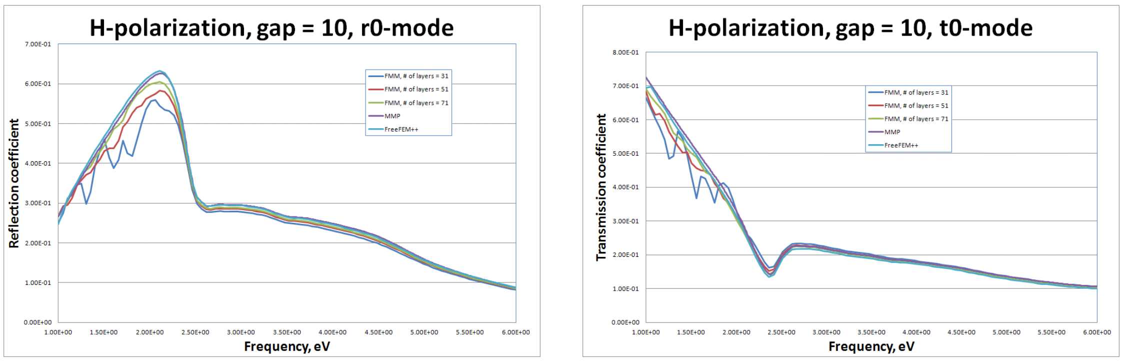

Figure 12: Comparison of the FMM and FEM results

From this gauging model test we conculde that FreeFem++ produces satisfactory (*for this particular case study*) results and we can proceed to our final goal – calculation of the whole model.

7.2. Results for the Solar-Cell-Model

We go back to the original model in question (fig.13) and compare two results:

– FMM only;

– combined FMM and FEM method.

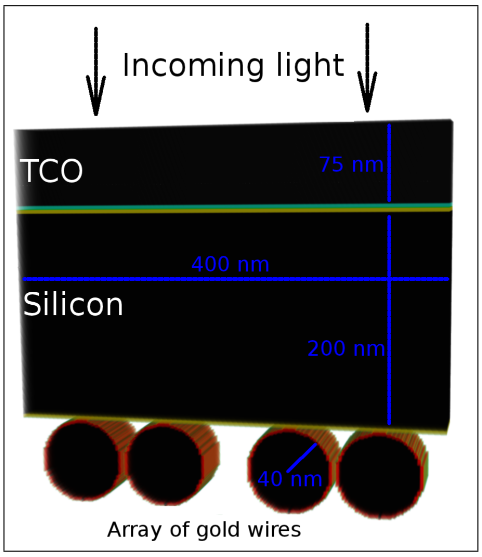

Figure 13: Test model

The TCO and silicon layers are exactly the same in both cases. The golden wire-array model in FEM has been described above in detail. The FMM model of this layer has 41 layers.

The resulting plots for reflection and transmission are presented on the fig.14.

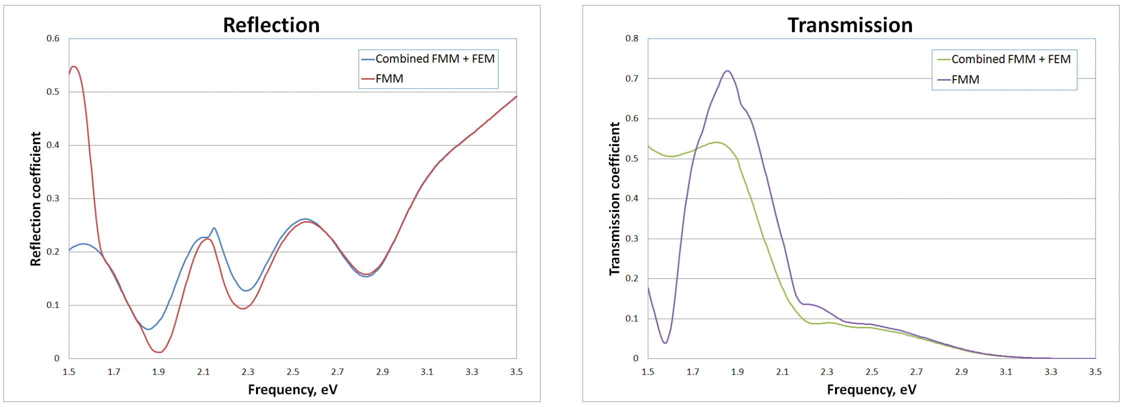

Figure 14: Comparison of the FMM and FMM+FEM results

As we can see the results differ a lot. But we have anticipated it (see fig.12) and we actually hoped to see this kind of difference.

Due to our tests on the gauging model we are sure that S-matrices created using FEM (i.e. FreeFem++) are proper, thus we can conclude that the combined-method approach gives more reasonable results than FMM.

8. Conclusion

We have shown that using combined-method approach for solving numerically Maxwell equations works well, moreover it produces more reasonable results than FMM only and does not produce oscillations common to the latter.

Of course, calculation of the S-matrix with FEM is very time consuming, but once a scattering matrix of the structure is calculated it can be used as a part of more complex structures. Thus it can be used many times in different models. Besides the calculation of the matrix with FEM could be accelerated via modes reduction, i.e. for FEM part we could take less modes in S-matrix than for FMM part, and fill the gaps with zeros, since for FEM the precision of the calculation does not depend on the number of modes.

Appendix A. FreeFem++ model code

```
//-------------------------------------------------------------------------------
//we leave out the creation of geometry and mesh
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
//constants definition
//-------------------------------------------------------------------------------

real c          =   2.99792458e10;
real eVrads     =   1.5192742e15;
real sq         =   1e-14;
real li         =   1e-7;
real pii        =   3.1415926535;

//input section
real omega;
cout << "Input omega" << endl;
cin >> omega;
real coof       =   omega * omega / (c * c) * (eVrads * eVrads);

real epsDsilver;
cout << "Input epsDsilver" << endl;
cin >> epsDsilver;

int N;
cout << "Input number of modes to calculate" << endl;
cin >> N;

//-------------------------------------------------------------------------------
//  functions description
//-------------------------------------------------------------------------------

func complex pmlS (real x, real y, real init)
{
complex pmlSmax = 40.0i;
return  init + init * ( pmlSmax * ( sizePML - y ) / sizePML ) * real( y < sizePML )
             + init * ( pmlSmax * ( y - sizeTOP + sizePML )/ sizePML )
                    * real( y > sizeTOP - sizePML );
}

func complex eps(real x, real y)
{
return      1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
    ( y < sizeTOP/2.0 - sizeCYLINDER/2.0 ) )
   +  1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
                  ( y > sizeTOP/2.0 + sizeCYLINDER/2.0 ) )
   +  1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
    ( y >= sizeTOP/2.0 - sizeCYLINDER/2.0 ) *
    ( y <= sizeTOP/2.0 + sizeCYLINDER/2.0 ) )
        + (epsDsilver - 1.0) * real( (y - sizeTOP / 2.0)^2 +
                                 (x - (sizeHORIZ + sizeGAPhor + sizeCYLINDER) / 2.0)^2
```

```
                                      <= (sizeCYLINDER/2.0)^2 )
   + (epsDsilver - 1.0) * real( (y - sizeTOP / 2.0)^2 +
      (x - (sizeHORIZ - sizeGAPhor - sizeCYLINDER) / 2.0)^2
      <= (sizeCYLINDER/2.0)^2 )
         + pmlS(x,y,1.0)      * real( ( y < sizePML ) + (y > sizeTOP - sizePML ) );
}

func complex epsi(real x, real y)
{
return      1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
( y < sizeTOP/2.0 - sizeCYLINDER/2.0 ) )
   +  1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
    ( y > sizeTOP/2.0 + sizeCYLINDER/2.0 ) )
   +  1.0 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) *
    ( y >= sizeTOP/2.0 - sizeCYLINDER/2.0 ) *
    ( y <= sizeTOP/2.0 + sizeCYLINDER/2.0 ) )
         + (epsDsilver - 1.0) * real( (y - sizeTOP / 2.0)^2 +
            (x - (sizeHORIZ + sizeGAPhor + sizeCYLINDER) / 2.0)^2
            <= (sizeCYLINDER/2.0)^2 )
   + (epsDsilver - 1.0) * real( (y - sizeTOP / 2.0)^2 +
      (x - (sizeHORIZ - sizeGAPhor - sizeCYLINDER) / 2.0)^2
      <= (sizeCYLINDER/2.0)^2 )
         + 1.0 / pmlS(x,y,1.0) * real( ( y < sizePML ) + (y > sizeTOP - sizePML ) );
}

func complex muyi(real x, real y)
{
return   1.0                 * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) )
       + pmlS(x,y,1.0)       * real( ( y < sizePML ) + ( y > sizeTOP - sizePML ) );
}

func complex muxi(real x, real y)
{
return     1.0                   * real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) )
       + ( 1.0 / pmlS(x,y,1.0) ) * real( ( y < sizePML ) + ( y > sizeTOP - sizePML ) );
}

func complex Einc(real x, real y, real kx, real ky, int flag)
{

if ( ky >= 0)
{
return exp( ( kx * x + flag * ky * (y  + (flag - 1 ) *
         sizeTOP / 2 - flag * (sizePML + sizeAIR)) ) * (1i) ) *
         real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) );
}
else
{
return exp( ( kx * x + flag * abs(ky) * (y + (flag - 1 ) *
  sizeTOP / 2 - flag * (sizePML + sizeAIR)) * (1i)) * (1i) ) *
  real( ( y >= sizePML ) * ( y <= sizeTOP - sizePML ) );
}
return 0;
}
```

```
//-------------------------------------------------------------------------------
//  weak formulations
//-------------------------------------------------------------------------------

real kxT, kyT, kySQ;
real k0 = omega / c * eVrads * li;
complex kyC;
int flagT;

fespace Vh(Th, P2, periodic=[[102, y], [104, y]]);
Vh<complex> E, v, Etot;
Vh Eabs, vabs;
fespace Vhvec(Th, [P2,P2], periodic=[[102, y], [104, y]]);
Vhvec<complex> [Ex, Ey], [vx, vy];

problem HELMHOLZE(E,v) =
int2d(Th)(muyi(x,y) * dx(E) * dx(v) + muxi(x,y) * dy(E) * dy(v))
- int2d(Th)( v * coof * E * eps(x,y) * sq)
+ int2d(Th)( v * coof * Einc(x,y,kxT,kyT,flagT) * ( 1.0 - eps(x,y) ) * sq)
+ on(101, E=0)
+ on(103, E=0);

problem HELMHOLZH([Ex,Ey],[vx,vy]) =
int2d(Th)(muxi(x,y) * dy(Ex) * dy(vx) - muxi(x,y) * dx(Ey) * dy(vx))
- int2d(Th)( vx * coof * Ex * eps(x,y) * sq)
+ int2d(Th)( vx * coof * kyC / k0 * Einc(x,y,kxT,kyT,flagT) * ( 1.0 - eps(x,y) ) * sq)
+ int2d(Th)(muxi(x,y) * dx(Ey) * dx(vy) - muxi(x,y) * dy(Ex) * dx(vy))
- int2d(Th)( vy * coof * Ey * epsi(x,y) * sq)
+ int2d(Th)( vy * coof * -kxT / k0 * Einc(x,y,kxT,kyT,flagT) * ( 1.0 - epsi(x,y) ) * sq)
+ on(101, Ex=0)
+ on(103, Ex=0)
+ on(101, Ey=0)
+ on(103, Ey=0);

problem TOTE(Etot,v) = int2d(Th)(Etot * v) -
   int2d(Th)(Einc(x,y,kxT,kyT,flagT) * v) -
   int2d(Th)(E * v);

problem TOTH(Etot,v) = int2d(Th)(Etot * v) -
   int2d(Th)(Einc(x,y,kxT,kyT,flagT) * v) -
   int2d(Th)(Ex * k0 / kyC * v);

problem EXTOE(E,v)   = int2d(Th)(E * k0 / kyC * v) - int2d(Th)(Ex * v);

int ifoutput = 1;

//-------------------------------------------------------------------------------
//  parameters definition
//-------------------------------------------------------------------------------

for ( int i = 0; i < 4*N+2; i = i + 1)
{
if (i == 0 || i == 2*N+1)
```

```
{
kxT = 0; kyT = omega / c * eVrads * li; kyC = kyT; flagT = (-1)^i;
}
else
{
if ( i > 0 && i < 2 * N + 1)
{
kxT = ((-1)^((i+1)%2)) * floor((i+1)/2) * (2 * pii / sizeHORIZ);
flagT = 1;
}
if ( i > 2 * N + 1 && i < 4 * N + 2)
{
kxT = ((-1)^(i%2)) * floor((i - 2*N)/2) * (2 * pii / sizeHORIZ);
flagT = -1;
}
kySQ = (omega / c * eVrads * li) * (omega / c * eVrads * li) - kxT * kxT;
if (kySQ >= 0)
{
kyT = sqrt( kySQ);
kyC = kyT;
}
else
{
kyT = -1 * sqrt( abs(kySQ) );
if (flagT == 1)
{
kyC = kyT * -1.0i;
}
else
{
kyC = kyT * 1.0i;
}
}
}

//-----------------------------------------------------------------------------
//  solving equations
//-----------------------------------------------------------------------------

HELMHOLZH;
TOTH;
EXTOE;

//-----------------------------------------------------------------------------
//  output
//-----------------------------------------------------------------------------

if (ifoutput) {
cout << "case: kx = " << kxT << "; ky = " << kyT << "; flag = " << flagT << endl;

{
cout << "results on the bottom line: gap = " << sizeGAP << endl;
for (int i = 0; i < Th.nt; i++)
{
```

```
for (int j=0 ; j <3; j++)
{
if (flagT == 1) {
if ( Th[i][j].y == sizePML + sizeAIR - sizeGAP)
cout << Th[i][j].x << " "
 << real(E[][Vh(i,j)]) << " "
 << imag(E[][Vh(i,j)]) << endl;
}
else {
if ( Th[i][j].y == sizePML + sizeAIR - sizeGAP)
cout << Th[i][j].x << " "
 << real(Etot[][Vh(i,j)]) << " "
 << imag(Etot[][Vh(i,j)]) << endl;
}
}
}
cout << "end of line" << endl;
}

{
cout << "results on the top line: gap = " << sizeGAP <<endl;
for (int i = 0; i < Th.nt; i++)
{
for (int j=0 ; j <3; j++)
{
if (flagT == 1) {
if ( Th[i][j].y == sizePML + sizeAIR + sizeCYLINDER + sizeGAP)
cout << Th[i][j].x << " "
 << real(Etot[][Vh(i,j)]) << " "
 << imag(Etot[][Vh(i,j)]) << endl;
}
else {
if ( Th[i][j].y == sizePML + sizeAIR + sizeCYLINDER + sizeGAP)
cout << Th[i][j].x << " "
 << real(E[][Vh(i,j)]) << " "
 << imag(E[][Vh(i,j)]) << endl;
}
}
}
cout << "end of line" << endl;
}

cout << "end of case: kx = " << kxT << "; ky = " << kyT <<  "; flag = " << flagT << endl;
}
}
```

Appendix B. C++ code

```
#include <iostream>
#include <cstdlib>
#include <ccomplex>
#include <cmath>
#include <fstream>
#include <vector>
#include <string>
#include <sstream>
#include <cassert>
#include <algorithm>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip>

#include "cmd_flags.h"
#include "fftw3.h"

static const double PI        = 3.141592653589793238462643383279;
static const double c         = 2.99792458e10;
static const double eVrads    = 1.5192742e15;
static const double li        = 1e-7;

double gap_bottom, gap_top, omega, period;

using namespace std;

struct Complex {
double re, im;
Complex(double r, double i): re(r), im(i) {}
Complex(): re(0), im(0) {}
};

ostream& operator <<(ostream& os, const Complex& x) {
return os << setw(15) << fixed << setprecision(6) << x.re
   << setw(15) << fixed << setprecision(6) << x.im << endl;
}

struct DataPoint {
double x, y;
Complex v;
};

void StripWhitespaces(string* s, const string& whitespaces = " \t\n\r") {
  const size_t start_pos = s->find_first_not_of(whitespaces);
  if (start_pos == string::npos) {
    s->clear();
    return;
  }
  const size_t end_pos = s->find_last_not_of(whitespaces);
  *s = s->substr(start_pos, end_pos - start_pos + 1);
}
```

```
struct CaseData {
double kx, ky;
vector<DataPoint> top, bottom;
};

class LineReader {
public:
LineReader(fstream &is) : is_(is), line_counter_(0) {}

bool Done() const {return !is_.is_open() || is_.eof();}

string GetLine() {
string line;

while (line.empty() && !Done()) {
++line_counter_;
getline(is_, line);
StripWhitespaces(&line);
// Skip comments and empty lines.
if (HasPrefix(line, "#") ||
HasPrefix(line, "%") ||
HasPrefix(line, "//"))  line.clear();
}
return line;
}

void Error (const string& s) {
cerr << "ERROR at line " << line_counter_ << ": " << s << endl;
exit(-1);
}

private:
fstream& is_;
int line_counter_;
};

struct DataPointLess{
bool operator()(const DataPoint& p1, const DataPoint& p2) { return p1.x < p2.x; }
};

void ReadPointValues (LineReader* lr, vector<DataPoint>* points) {
points->clear();
DataPoint p;

while ( !lr->Done() ) {
string line = lr->GetLine();
if ( HasPrefix(line,"end of line") ) break;

istringstream line_stream(line + " ");
line_stream >> p.x >> p.v.re >> p.v.im;

if (line_stream.eof()) lr->Error("Unsupported format");

points->push_back(p);
```

```
}
sort(points->begin(), points->end(), DataPointLess());

int pos = 1;
for ( int i = 1; i < points->size(); ++i ) {
if (points->at(i).x > points->at(i - 1).x + 1e-12) {
points->at(pos) = points->at(i);
++pos;
}
}
points->resize(pos);
}

void ReadCaseData (LineReader* lr, CaseData* cd) {

string line;

if (lr->Done()) lr->Error("Expecting line: 'results on the bottom line'");
line = lr->GetLine();
sscanf(line.c_str(), "results on the bottom line: gap = %lf", &gap_bottom);
ReadPointValues(lr, &cd->bottom);

if (lr->Done()) lr->Error("Expecting line: 'results on the top line'");
line = lr->GetLine();
sscanf(line.c_str(), "results on the top line: gap = %lf", &gap_top);
ReadPointValues(lr, &cd->top);

if (cd->bottom.size() != cd->top.size()) {
for (int k = 0; k < cd->bottom.size(); ++k){
cerr << "Bottom [" << k << "] = " << cd->bottom[k].x << endl;
}

for (int k = 0; k < cd->top.size(); ++k){
cerr << "Top [" << k << "] = " << cd->top[k].x << endl;
}

lr->Error("Bottom/Top length mismatch");
}
}

void ReadAllCases(fstream& is, vector<CaseData>* cases) {
  cases->clear();

  string line;
  LineReader lr(is);

  while ( !lr.Done() ) {
line = lr.GetLine();
if ( HasPrefix(line,"case: ") ) {
CaseData cd;
sscanf(line.c_str(), "case: kx = %lf; ky = %lf", &cd.kx, &cd.ky);
ReadCaseData(&lr, &cd);
cases->push_back(cd);
}
```

```
  }
}

void FFT (const vector<DataPoint>& input, vector<Complex>* output) {
    int N = input.size() - 1;
    fftw_complex *in, *out;
    fftw_plan p;
    int i;

    in = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*N);
    out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*N);

p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_MEASURE);
for (int k = 0; k < N; ++k) {
in[k][0] = input[k].v.re;
in[k][1] = input[k].v.im;
}

    fftw_execute(p);
    // generate output with right order of modes
    output->clear();
    for (int k = 0; k < N; ++k){
      if (k % 2 == 1||k == 0){
i = (k + 1) / 2;
      }
      else{
i = N - k / 2;
      }
      output->push_back(Complex(out[i][0]/N, out[i][1]/N));
    }
    //free variables
    fftw_destroy_plan(p);
    fftw_free(in);
    fftw_free(out);
}


void usage() {

cout << "usage: \n"
 << "case1: using existing file with FreeFem output \n"
 << "  e.g. FFT.exe --log=output.dat \n"
 << "case2:  run FreeFem model for specified parameters\n"
 << "  e.g. FFT.exe --model=model_01_010412.cpp --omega=1 --N=15\n"
 << endl;

exit(-1);
}

void RunModel (const string& FreeFem_model, double omega, int N,
               vector<CaseData>* cases, string conf_name, string number) {
static string kFreeFemInputFilename = conf_name + "FreeFem_input_" + number + ".dat";
static string kFreeFemOutputFilename = conf_name + "FreeFem_output_" + number + ".log";
fstream FreeFem_input;
```

```
FreeFem_input.open(kFreeFemInputFilename, fstream::out);
FreeFem_input << omega << endl;
FreeFem_input << N << endl;

string cmd = "FreeFem++ " + FreeFem_model + " <" +
                     kFreeFemInputFilename + " >" + kFreeFemOutputFilename;
system(cmd.c_str());

fstream FreeFem_output;
FreeFem_output.open(kFreeFemOutputFilename, fstream::in);
if (!FreeFem_output.is_open()) {
cerr << "cannot open " << kFreeFemOutputFilename << endl;
exit(-1);
}

ReadAllCases(FreeFem_output, cases);
FreeFem_output.close();
}

void OutputResultsTxt ( const vector<Complex>& results, ostream& os ) {

int N = sqrt(results.size() + 0.5);
assert ( N * N == results.size() );

os << N << endl;
for (int i = 0; i < results.size(); ++i)  {
if ( i %  N == 0 ) os << endl;
os << results[i];
}
}

void OutputResultsBin ( const vector<Complex>& results, const string& filename) {

fstream f;
f.open(filename, fstream::out | fstream::binary);
f.clear();
int N = sqrt(results.size() + 0.5);
assert ( N * N == results.size() );

//write zeros to the second polarization
int i, ii, it, ikx;
double z = 0.0;
int NN = N / 2;
assert ( NN * 2 == N );
int NNN = ( NN - 1 ) / 2;
assert ( 2 * NNN + 1 == NN );
double ktot = omega / c * eVrads * li;
double gap, kx, val_re, val_im, ky, scale;

for (i = 0; i < results.size(); ++i) {
  //transpose matrix
  it = (i - (i / N) * N) * N + i / N;
  //phase correction
  ii = i / N;
  if (ii < NN) {gap = gap_top;}
  else {gap = gap_bottom; ii = ii - NN;}
  ikx = (ii + 1) / 2;
  kx = ikx * 2 * PI / period;
  ky = ktot * ktot - kx * kx;
  if (ky >= 0){
    ky = sqrt(ky);
    val_re = results[it].re * cos(ky * gap) + results[it].im * sin(ky * gap);
    val_im = results[it].im * cos(ky * gap) - results[it].re * sin(ky * gap);
  }
  else{
    scale = exp(sqrt(-ky) * gap);
    val_re = scale * results[it].re;
    val_im = scale * results[it].im;
  }
  //write element
  f.write((char*) (&val_re), sizeof(z));
  f.write((char*) (&val_im), sizeof(z));
if ((i+1) % NN == 0){
  for (ii = 0; ii < NN; ++ii){
    f.write((char*) (&z), sizeof(z));
    f.write((char*) (&z), sizeof(z));
  }
}
if ((i+1) % (N * NN) == 0){
  for (ii = 0; ii < NN * NN * 4; ++ii){
    f.write((char*) (&z), sizeof(z));
    f.write((char*) (&z), sizeof(z));
  }
}
}
f.close();
}

void OutputResults (const vector<CaseData>& cases, const string& bin_filename,
         const string& text_filename) {

int N = (cases.size() -2 ) / 4;
assert (4 * N + 2 == cases.size());

vector<Complex> output_FFT;

for (int i = 0; i < cases.size(); ++i) {
vector<Complex> out;
const CaseData& cd = cases[i];
FFT(cd.top, &out);

assert (out.size() >=2 * N + 1);

for (int j = 0; j < 2 * N + 1; ++j)  output_FFT.push_back(out[j]);

FFT(cd.bottom, &out);

assert (out.size() >=2 * N + 1);
```

```
for (int j = 0; j < 2 * N + 1; ++j) output_FFT.push_back(out[j]);
}

if ( !text_filename.empty() ) {
fstream text_out;
text_out.open(text_filename, fstream::out);
OutputResultsTxt(output_FFT, text_out);
text_out.close();
}

if ( !bin_filename.empty() ) OutputResultsBin (output_FFT, bin_filename);
else { cerr << "Bin_filename not specified!" << endl; }
}

int main (int argc, char* argv[])
{
vector<CaseData> cases;

string model_filename;
string text_output_filename;
string bin_output_filename;
string conf_name;

int N = 0;
string number;
string count;

GetFlagValue(argc, argv, "model", &model_filename);
GetFlagValue(argc, argv, "text_output", &text_output_filename);
GetFlagValue(argc, argv, "bin_output", &bin_output_filename);
GetFlagValue(argc, argv, "conf_name", &conf_name);

GetFlagValue(argc, argv, "omega", &omega);
GetFlagValue(argc, argv, "period", &period);
GetFlagValue(argc, argv, "N", &N);
GetFlagValue(argc, argv, "number", &number);
GetFlagValue(argc, argv, "count", &count);

if ( !text_output_filename.empty() && !conf_name.empty() ) {
  text_output_filename = conf_name + text_output_filename;
}
if ( !bin_output_filename.empty() && !conf_name.empty() ) {
  bin_output_filename = conf_name + bin_output_filename;
}

cout << "\r" << count;
RunModel(model_filename, omega, N, &cases, conf_name, number);
OutputResults (cases, bin_output_filename, text_output_filename);
}
```

References

[1] Lifeng Li, *New formulation of the Fourier modal method for crossed surface-relief gratings*. Journal of the Optical Society of America A, Vol.14, No.10, Oct 1997.

[2] *FreeFem++ Manual*. http://www.FreeFem.org/.

[3] Jianming Jin, *The Finite Element Method in Electromagnetics*. A Wiley-Interscience Publication, 2002.