Cálculo de las corrientes de Foucault utilizando elementos finitos hp de Nedeléc

Juan Sebastián Flórez Jiménez[1, a)]
Departamento de Física, Universidad Nacional de Colombia, Bogotá, Colombia

I. FORMULACIÓN DEL PROBLEMA

A. Ecuaciones de Maxwell

Las cantidades físicas que se involucran en un problema de electromagnetismo son el campo eléctrico E, el campo magnético H, la inducción magnética B, la inducción eléctrica D, y la densidad de corriente J. La relación entre la inducción eléctrica y el campo eléctrico se pueden hallar suele plantearse como lineal, sin embargo, es posible utilizar formulaciones no lineales. En el caso lineal la inducción eléctrica se expresa como $D = \epsilon E$ y la inducción magnética como $B = \mu H$; μ y ϵ son tensores de segundo rango (matrices) que pueden depender de la posición y del tiempo.

Los tensores ϵ y μ dependen de las propiedades microscópicas de los materiales, ya que dan cuenta de la respuesta del material ante campos externos. Debido a que la información con la cual se cuenta son los datos experimentales de estas cantidades, y tales datos corresponden a variables macroscópicas, la primera aproximación que se hace es considerar el medio como isotrópico, con lo cual las expresiones para la inducción magnética y eléctrica son:

$$D = \epsilon(r)E,\ B = \mu(r)H$$

Sea ρ la densidad de carga libre, entonces, el sistema de ecuaciones de Maxwell se escribe como:

$$\begin{cases} \dfrac{\partial \mathcal{D}}{\partial t} + \mathcal{J} = \operatorname{curl} \mathcal{H} & \text{Maxwell–Ampère equation} \\ \dfrac{\partial \mathcal{B}}{\partial t} + \operatorname{curl} \mathcal{E} = 0 & \text{Faraday equation} \\ \operatorname{div} \mathcal{D} = \rho & \text{Gauss electrical equation} \\ \operatorname{div} \mathcal{B} = 0 & \text{Gauss magnetic equation ,} \end{cases}$$

Figura 1. Sistema de ecuaciones de Maxwell[1]

Se añade otra variable al sistema de ecuaciones, la conductividad (σ), la cual permite expresar la densidad de corriente en función del campo eléctrico:

$$J = \sigma E + J_e$$

La anterior ecuación J_e da cuenta de las fuentes externas de las fuentes externas de corriente eléctrica, y se supone que $div(J_e) = 0 \partial \rho_e / \partial t$, i.e. las fuentes externas no generan cambios en la distribución de la densidad de carga.

La formulación presentada es dependiente del tiempo, sin embargo, en varias aplicaciones prácticas se utilizan campos periódicos en el tiempo:

$$E(x,t) = Re[\overline{E}(x)e^{i\omega t}],\ H(x,t) = Re[\overline{H}(x)e^{i\omega t}]$$

$$J_e = Re[J_* e^{i\omega t + \phi}] = Re[\overline{J_e} e^{i\omega t}]$$

Utilizando estas expresiones se pueden plantear las ecuaciones de Maxwell armónicas, las cuales utilizan variables complejas i.e. los campos H, E, y J_e son campos vectoriales complejos.

$$\begin{cases} \operatorname{curl} \mathbf{H} - (i\omega \boldsymbol{\varepsilon} + \boldsymbol{\sigma})\mathbf{E} = \mathbf{J}_e \\ \operatorname{curl} \mathbf{E} + i\omega \boldsymbol{\mu} \mathbf{H} = \mathbf{0} \ , \end{cases}$$

Figura 2. Ecuaciones de Maxwell armónicas[1]

Tal que la densidad de carga se calcula en cada material a partir del valor del campo eléctrico y la permitividad eleć trica (ϵ):

$$\rho(\mathbf{x}, t) = \operatorname{div}\left(\boldsymbol{\varepsilon}(\mathbf{x})\mathcal{E}(\mathbf{x}, t)\right) = \operatorname{div}\left(\operatorname{Re}[\boldsymbol{\varepsilon}(\mathbf{x})\mathbf{E}(\mathbf{x})e^{i\omega t}]\right)$$

Figura 3. Fórmula para la densidad de carga[1]

B. Corrientes de Foucault

La ecuación de Faraday postula que un cambio del campo magnético en el tiempo produce un campo eléctrico, por lo tanto, dentro de un conductor se produce una corriente de Foucault o también conocida como Eddy current $J_{eddy} = \sigma E$. Esta corriente disipa energía en forma de calor, lo cual es conocido como el efecto Joule, y es ampliamente utilizado en la industria metalúrgica para fundir metales. La presencia de las corriente de Foucault también puede producir efectos indeseados, como el sobre-calentamiento de dispositivos

[a)] jsflorezj@unal.edu.co. c.c 1032469056

electrónicos, o la pérdida de energía. Adicionalmente, se producen corrientes de Foucault dentro del cuerpo de los pacientes sometidos a tomografía de inducción magnética (MIT, Magnetic Induction Resonance), cuyos efectos pueden ser detectado y analizados para obtener información sobre los tejidos internos del paciente[2].

Si se supone que los campos tienen frecuencias bajas (longitud de onda grande respecto de las distancias características del sistema), se pueden despreciar las corrientes de desplazamiento i.e. $\partial D/\partial t = i\omega\epsilon E \approx 0$[1]. Es así como se obtienen las ecuaciones magnéticas cuasi-estáticas de las ecuaciones de Maxwell, también conocidas como eddy current aproximation.

$$\begin{cases} \operatorname{curl}\mathbf{H} - \boldsymbol{\sigma}\mathbf{E} = \mathbf{J}_e & \text{in } \Omega \\ \operatorname{curl}\mathbf{E} + i\omega\boldsymbol{\mu}\mathbf{H} = \mathbf{0} & \text{in } \Omega \\ \operatorname{div}(\boldsymbol{\varepsilon}\mathbf{E}) = 0 & \text{in } \Omega_I \end{cases}$$

Figura 4. Aproximación magneto-estática de las ecuaciones de Maxwell armónicas[1]

Donde Ω denota el dominio considerado y Ω_I la parte del dominio que está ocupado por medios aislantes; ya que la corriente de desplazamiento no se incluye en esta formulación es necesario imponer que no hay acumulación de carga dentro del aislante i.e. $div(\epsilon E) = 0$. Además, se supone que no hay corrientes dentro de carga dentro de los aislantes:

$$\operatorname{div}\mathbf{J}_e = 0 \quad \text{in } \Omega_I$$

C. Potencial vectorial

Ya que la divergencia de B es nula, se puede escribir como el rotacional de un campo vectorial, el cual es llamado potencial vectorial A. Sin embargo, el potencial vectorial A no es único, pues si se suma la divergencia de un campo escalar a A se recupera el mismo campo vectorial B. El hecho de que A no sea único implica que existe un grado de libertad adicional por tener en cuenta al solucionar el sistema de ecuaciones de Maxwell, que corresponde al campo eléctrico. Es por esto que el potencial vectorial se define como $A + \nabla\phi$, donde A es una solución de la ecuación magnética de Gauss, y ϕ es un potencial que se ajusta para cumplir la ecuación:

$$E = -\nabla\phi - \frac{\partial A}{\partial t}$$

Aplicando este formalismo a las ecuaciones magnéticas cuasi-estáticas el sistema de ecuaciones se reduce:

$$\begin{cases} curl(\mu^{-1}curl(A)) - \sigma\left(-\nabla\phi - i\omega A\right)) = J_e \\ div(\epsilon E) = 0,\ in\ \Omega_I \end{cases}$$

En el régimen cuasi-estático se puede despreciar la contribución de $\nabla\phi$, con lo cual se obtiene una sola ecuación diferencial con restricciones en ciertas partes del dominio (medio no conductor)[2]. La forma débil de este problema diferencial se expresa como:

$$\int_\Omega \mu_r \operatorname{curl}\boldsymbol{A}\cdot\operatorname{curl}\boldsymbol{w}\mathrm{d}\Omega + \int_\Omega \tilde{\kappa}\boldsymbol{A}\cdot\boldsymbol{w}\mathrm{d}\Omega = \mu_0\int_{\Omega_{NC}}\boldsymbol{J}^s\cdot\boldsymbol{w}\mathrm{d}\Omega \qquad \forall\boldsymbol{w}\in\boldsymbol{V},$$

Done el parámetro $\widetilde{\kappa}$ toma diferentes valores dentro del conductor y fuera de este. Además, en el medio no conductor se utiliza un parámetro de regularización para definir $\widetilde{\kappa}$, porque la solución diverge en la interfaz entre conductor y no conductor si $\widetilde{\kappa} = 0$.

$$\widetilde{\kappa}\begin{cases} i\omega\mu_0\sigma,\ in\ \Omega_C \\ i\varepsilon,\ in\ \Omega_I \end{cases}$$

El espacio de funciones de prueba utilizadas es:

$$\boldsymbol{V} := \{\boldsymbol{A}\in\boldsymbol{H}(\operatorname{curl}) : \boldsymbol{n}\times\boldsymbol{A} = \boldsymbol{0} \text{ on } \partial\Omega\}$$

Sea $\hat{k}\overrightarrow{x} = \overrightarrow{b}$ el sistema lineal producido al escoger un espacio finito de funciones, entonces, la matriz $\hat{k}$ se componen de una matriz de masa y una matriz del rotacional, y el vector $\overrightarrow{b}$ corresponde a la proyección de la densidad de corriente externa al conductor sobre las funciones base del espacio de dimensión finita.

II. ELEMENTOS FINITOS DE NEDELÉC

Los elementos finitos de Nedeléc se definen a partir de los vértices y caras de la celda fundamental. En 2D los elemento de Nedeléc se definen a partir de los vértices de la celda fundamental, y en 3D se añaden las caras de la celda fundamental. Estos elementos son ampliamente utilizados al resolver las ecuaciones de Navier-Stokes y las ecuaciones de Maxwell, debido a que tiene divergencia nula; esto implica que no generan fuentes artificiales.

Los elementos de Nedeléc son definidos a partir de una arista o una cara de la celda fundamental, si se utiliza una arista un elemento de orden p toma el valor de un polinomio de grado p cuando es evaluado en tal arista. Si son definidos a partir de una cara entonces se

anulan en todos los vértices de la celda fundamental, lo cual garantiza la continuidad tangencial de la solución.

Se utilizaron los elementos finitos de Nedeléc definidos sobre hexaedros, ya que deal.ii utiliza mallados hexaedricos. La implementación de estos elementos finitos fue creada por R.M. Kynch[2] a partir del desarrollo expuesto por Sabine Zaglmayr en su tesis doctoral[4].

Los elementos finitos de Nedeléc pertenecen al espacio $H(curl)$, por lo tanto, la norma utilizada para comparar diferentes soluciones es:

$$\|\boldsymbol{e}\|_{\boldsymbol{H}(\mathrm{curl})} := \left(\int_{\Omega} |\boldsymbol{e}|^2 + |\mathrm{curl}\ \boldsymbol{e}|^2 \mathrm{d}\Omega\right)^{1/2}$$

III. ESFERA CONDUCTORA DENTRO DE UNA CAMPO MAGNÉTICO UNIFORME ARMÓNICO

Se considera una esfera conductora de radio $R = 0{,}05$ m dentro de un campo magnético uniforme con flujo magnético $B_0 = \mu_0 H_0 = (0, 0, 1)^T$ y frecuencia angular $\omega = 100\pi rads^{-1}$. La conductividad de la esfera es $\sigma_C = 10^7 Sm^{-1}$ y permitividad magnética relativa $\mu_r = 20\mu_0$.

Se utiliza un archivo para definir el valor de las constante físicas a utilizar, tal que en éste se especifica el valor del parámetro de regularización usado al definir $\widetilde{\kappa}$, el tipo de variable utilizada para establecer el campo externo, las dimensiones del mallado a utilizar, etc.

Se genera una mallado para la esfera externa con radio 20 veces mayor al radio de la esfera conductora, lo cual permite despreciar la contribución de los campos de scattering sobre la frontera del mallado; los campos de scattering (producidos como respuesta al campo externo) decaen como $1/|x|$, por lo tanto, se pueden imponer condiciones de Dirichlet homogéneas para los campos de scattering si la frontera está lo suficientemente lejos del objeto dispersor[2].

IV. RESULTADOS

El problema planteado tiene solución teórica, la cual se encuentra implementada dentro del código. Se muestran las triangulaciones obtenidas, la solución numérica y la comparación de la solución numérica con la solución teórica. Para una triangulación con dos refinamientos globales y funciones base de segundo orden no se distingue la solución numérica de la solución teórica, lo cual si sucede si se utiliza un refinamiento global con funciones base de primer orden.

A. h=1,p=1,m=2

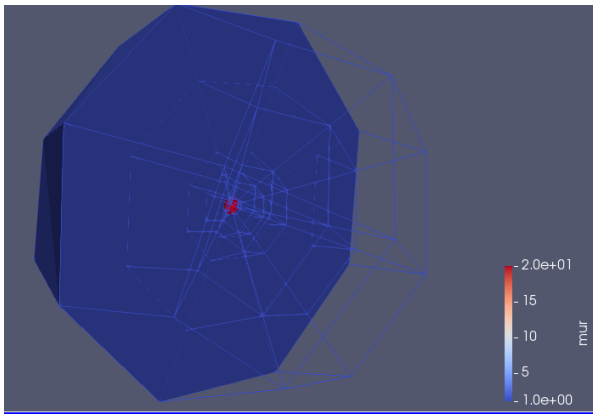

Figura 5. Triangulación con un refinamiento global

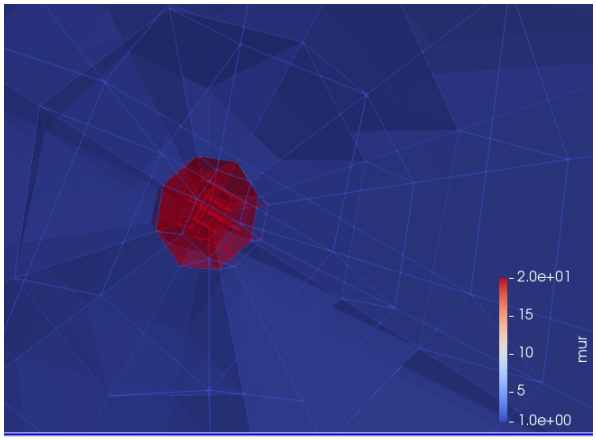

Figura 6. Triangulación con un refinamiento global

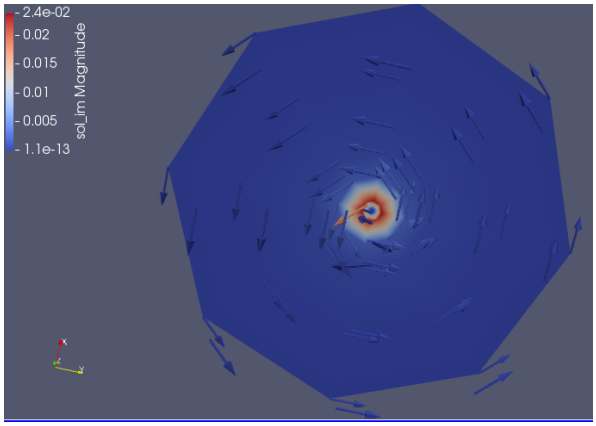

Figura 7. Solución numérica del campo eléctrico

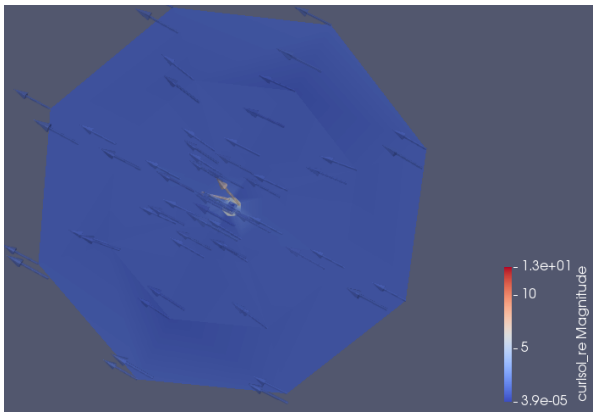

Figura 8. Solución numérica del campo magnético

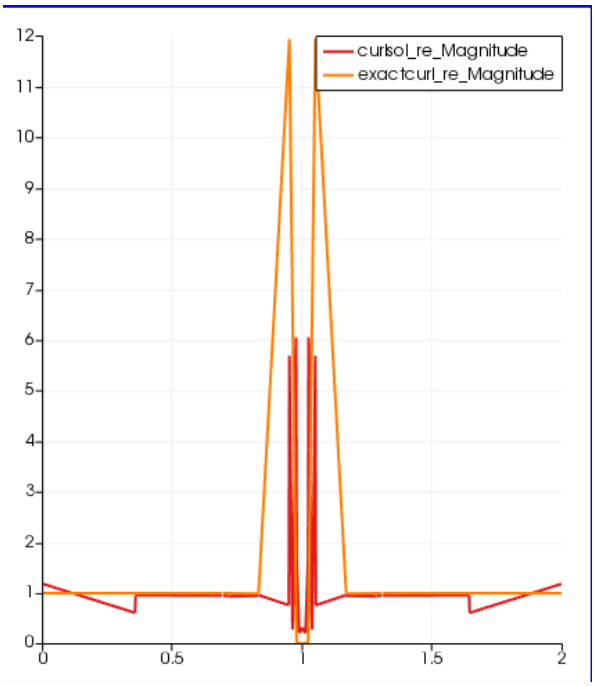

Figura 10. Comparación entre la solución numérica y la solución teórica del campo magnético sobre el eje x

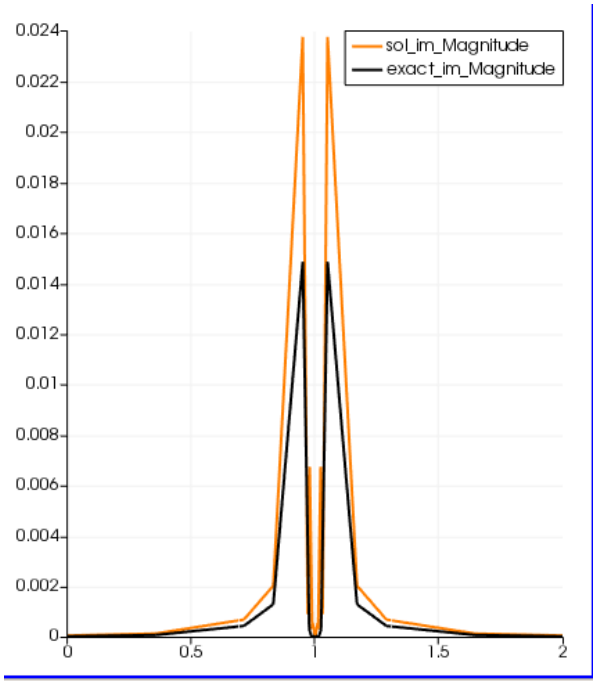

Figura 9. Comparación entre la solución numérica y la solución teórica del campo eléctrico sobre el eje x

B. h=2,p=2,m=2

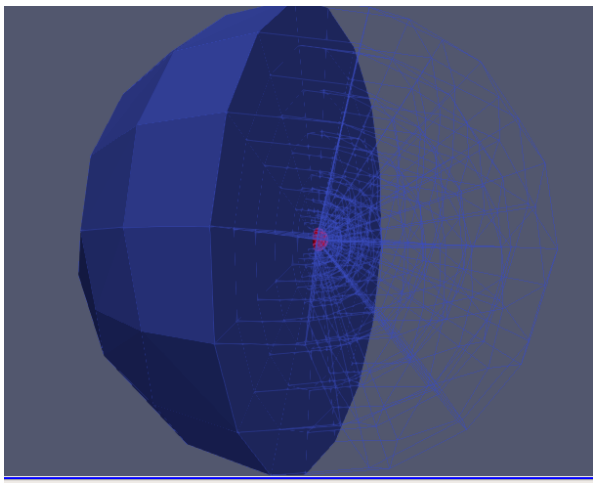

Figura 11. Triangulación con dos refinamientos globales

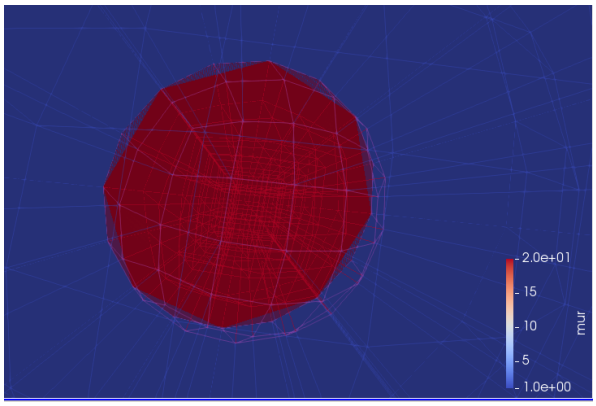

Figura 12. Triangulación con dos refinamientos globales

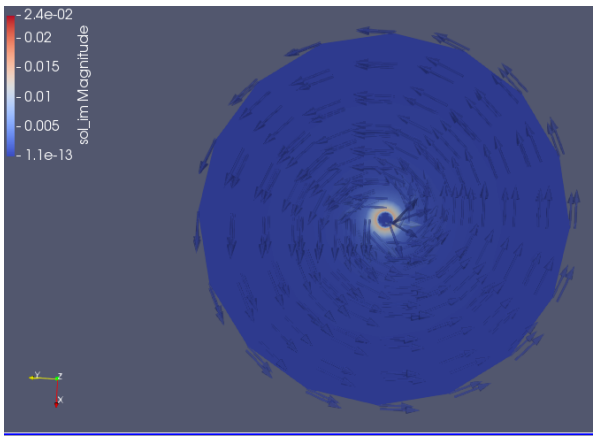

Figura 13. Solución numérica del campo eléctrico

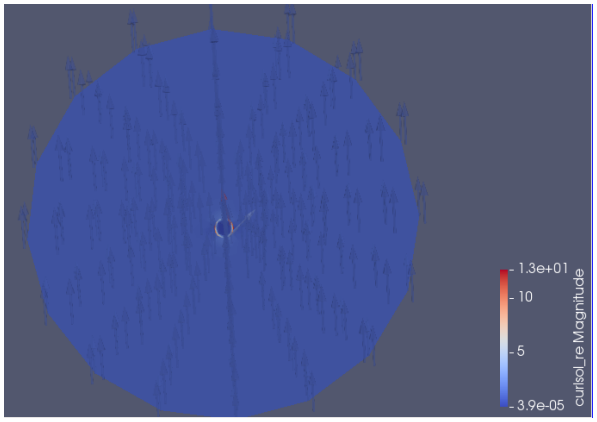

Figura 14. Solución numérica del campo magnético

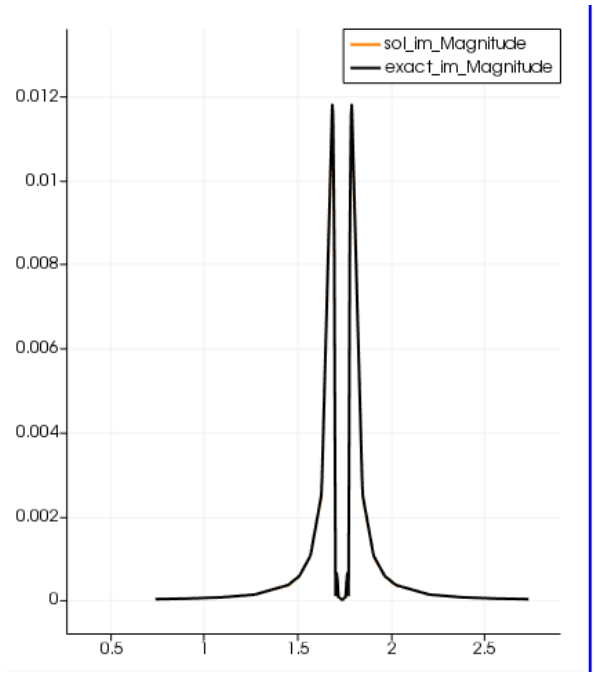

Figura 15. Comparación entre la solución numérica y la solución teórica del campo eléctrico sobre el eje x

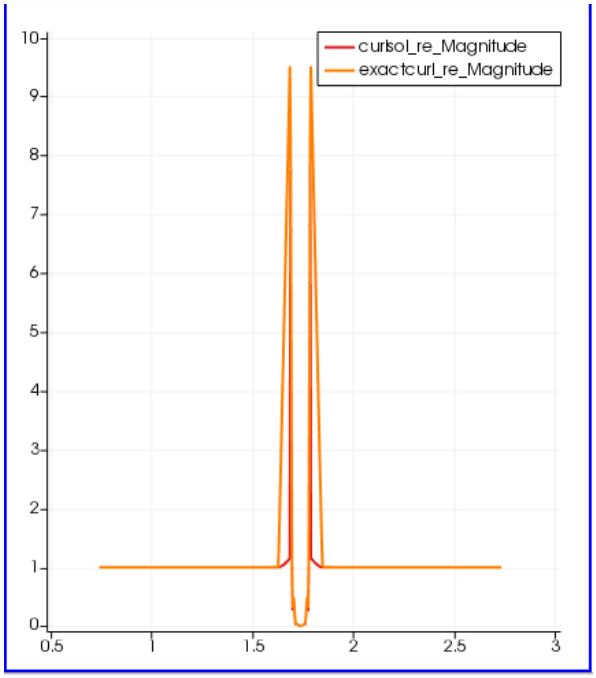

Figura 16. Comparación entre la solución numérica y la solución teórica del campo magnético sobre el eje x

REFERENCIAS

[1]Ana Alonso Rodríguez and Alberto Valli, .Eddy Current Approximation of Maxwell Equations Theory, algorithms and applications", Springer

[2]R.M. Kynch, P.D. Ledger, Resolving the sign conflict problem for hp–hexahedral Nédélec elements with application to eddy current problems",Computers and Structures 181 (2017) 41–54, http://dx.doi.org/10.1016/j.compstruc.2016.05.021
[3]Turner LR, Davey K, Emson CRI, Miya K, Nakata T, Nicolas A. Problems and workshops for eddy current code comparison. IEEE Trans Magnet 1988;24 (1):431–4
[4]Zaglmayr S, Schöberl J. High order Nédélec elements with local complete sequence properties. Int J Comput Math Electr Electron Eng (COMPEL) 2005;24:374–84.
[5]R.M. Kynch, P.D. Ledger. TBC
[6]P.D. Ledger, S. Zaglmayr, hp-Finite element simulation of three-diemensional eddy current problems on multiply connected domains. Computer Methods in Applied Mechanics and Engineering 199 (2010) 3386-3401.

V. CÓDIGO

El código utilizado fue creado por Ross Kynch y Paul Ledger como parte un trabajo de doctorado en el cual implementaron elementos hp-hexaédricos de Nedeléc para solucionar las ecuaciones magnéticas cuasi-estáticas de Maxwell. Este programa lo utilizaron para deducir las propiedades del medio a partir de los campos medidos cuando se realiza una MIT. El código creado es de libre acceso y se encuentra en un repositorio de github para aquellos que deseen leerlo y/o modificarlo: https://github.com/rosskynch/MIT_Forward, https://github.com/rosskynch/MIT_Inverse.

El código está creado a partir de deal.II, y consiste principalmente de dos conjuntos de archivos, los archivos que contienen los encabezados y declaraciones (.h) y los archivos que contienen el código fuente (.cc). En este se implementan la solución de cuatro problemas relacionados con corrientes de Foucault, los cuales fueron propuestos por el IEEE como parte del TEAM (Testing of Electromagnetic Analisys Methods) en 1988.

El código escrito por Ross Kynch y Paul Ledger implementa la solución de tres problemas pertenecientes al TEAM:

- Cascarón esférico metálico dentro de un campo magnético armónico
- Conductor en forma de toro dentro de un campo magnético armónico
- Plato conductor con un hueco asimétrico dentro de un campo magnético rotante.

El mallado del primer problema fue creado a partir de las herramientas de deal.II, sin embargo, el mallado para los otros dos problemas fue generado con el software *Cubit*. Por esto, se analizó la implementación del primer problema, dejando para trabajos posteriores los problemas donde se necesita crear un mallado externo.

Los archivos que componen el solucionador son:

- all_data.h
- backgroundfield.h
- eddycurrentfunction.h
- forwardsolver.h
- inputtools.h
- mydofrenumbering.h
- mypreconditioner.h
- myvectortools.h
- outputtools.h
- myfe_nedelec.h
- mynedelec_tools.h
- mypolynomials.h
- all_data.cc -
- backgroundfield.cc -
- eddycurrentfunction.cc -
- forwardsolver.cc -
- inputtools.cc-
- mydofrenumbering.cc-
- mypreconditioner.cc-
- myvectortools.cc
- outputtools.cc-
- myfe_nedelec.cc-
- mypolynomials.cc-

Los archivos listados cumplen las siguientes funciones:

- forwardsolver.cc/.h: implementa la clase .ᴱᵈdyCurent"la cual cuenta con funciones para ensamblar la matriz, el vector RHS, calcular las restricciones sobre los grados de libertad al imponer una divergencia nula de la inducción eléctrica en el medio no conductor, y solucionar el sistema lineal.
- backgroundfield.cc/.h: implementa funciones del tipo deal.ii que permitan evaluar las condiciones de frontera y crear las fuentes de corriente, como puede ser el caso de una bobina que genera el campo incidente.
- all_data.cc/.h: guarda los valores de los parámetros físicos del problema, como lo son el radio de la esfera conductora, la permitividad magnética de la esfera, el vector de polarización, la frecuencia de la onda incidente, etc.
- eddycurrentfunction.cc/.h:implementa una función de deal.ii que permite obtener el valor del campo en cualquier punto del mallado, al igual que el valor de su rotacional, y demás variables vectoriales de interés.

- inputtools.cc/.h: contiene funciones que permiten obtener datos de archivos externos y guardarlos en las variables internas del sistema e.g. el radio del mallado.
- mydofrenumbering.cc/.h: implementa funciones que se encargan de administrar la enumeración de los grados de libertad, de tal forma que no se presenten conflictos de signos en las fronteras de las celdas del mallado.
- myfe_nedelec.cc/.h: contiene las funciones asociadas con los elementos finitos propuestos por Sabine Zaglmayr en su tesis doctoral[4].
- mynedelec_tools.h: contiene las funciones que permiten calcular proyecciones utilizando las funciones base propuestas por Sabine Zaglmayr.
- outputtools.cc/.h: contienen las funciones que generan los archivos con la información de la solución del problema diferencial. El principal producto de la simulación es un archivo en formato vtk con la información de la triangulación, la solución numérica, la solución teórica, y la distribución espacial de las constantes materiales (permitividad y conductividad).
- mypreconditioner.cc/.h: implementa rutinas que permiten solucionar el el sistema lineal eficientemente, para más información consultar el artículo de Ross Kynch Resolving the sign conflict problem for hp–hexahedral Nédélec elements with application to eddy current problems"[2].
- mypolynomials.cc/.h: implementan funciones para calcular los coeficientes de los polinomios utilizados para definir las funciones base (polinomios de Legendre).
- myvectoortools.cc/.h: implementa funciones que permiten calcular el error de la solución numérica utilizando la norma asociada al espacio $H(curl)$ y calcular el valor del gradiente y el rotacional de la solución numérica.

El archivo que contiene la función "main().[es] "sphere_benchamark.ccz hace uso de todos los archivos mencionados anteriormente. Al compilar todos los archivos se crea un ejecutable, tal que es posible modificar el valor del grado de los polinomios de las funciones base, la cantidad de veces que se refina el mallado, y el orden del mapeo entre la celda unidad y las celdas del mallado.

Los comandos utilizados para compilar y correr el programa son:

```
$ cd benchmark/sphere_benchmark
$ mkdir build
$ cd build
$ cmake ../
$ make
$ ./sphere_benchmark -i ../input_files/input_filename -m i -h j -p k
```

El cual utiliza en archivo ïnput_filenameçomo fuente de los parámetros del problema, se genera un mapeo de orden i entre la celda referencia y las celdas del mallado, se realiza j veces un refinamiento del mallado y se utilizan elementos de Nedeléc de orden k. Se recomienda utilizar i=2, de otra forma se generarán más grados de libertad que los que realmente existen. El repositorio https://github.com/JuanSebastianFlorez/FEM-Eddy-Current-Application.git se encuentra el código utilizado en este trabajo, para replicar los resultados expuestos utilizar el archivo "sphere_team6_dealmesh.prmçomo archivo de entrada (input file).

Se recomienda tener la versión 8.3 de deal.ii actualizada al commit (SHA hash 79583e56..) del 6 de julio. La última versión de deal.ii (8.5) no es compatible con el código escrito por R.M. Kynch[2].

A. Archivo principal: sphere_benchmark.cc

Se crea la clase sphereBenchmark, la cual tiene funciones que utilizan los objetos definidos en los archivos que componen el solucionador.

★Encabezado del archivo, aquí se definen cuales librerías se utilizan y cuales son los archivos fuente con los encabezados de las funciones que se utilizan.

```
/*
 *    An MIT forward solver code based on the deal.II (www.dealii.org) library.
 *    Copyright (C) 2013-2015 Ross Kynch & Paul Ledger, Swansea Unversity.
 *
 *    This program is free software: you can redistribute it and/or modify
 *    it under the terms of the GNU General Public License as published by
 *    the Free Software Foundation, either version 3 of the License, or
 *    (at your option) any later version.
 *
 *    This program is distributed in the hope that it will be useful,
 *    but WITHOUT ANY WARRANTY; without even the implied warranty of
 *    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *    GNU General Public License for more details.
 *
 *    You should have received a copy of the GNU General Public License
 *    along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/tria_accessor.h>
#include <deal.II/grid/tria_iterator.h>
#include <deal.II/grid/tria_boundary_lib.h>

#include <deal.II/grid/manifold_lib.h>

#include <deal.II/fe/fe_nedelec.h>

//Archivos que componen este solucionador
#include <all_data.h>
#include <backgroundfield.h>
#include <eddycurrentfunction.h>
#include <forwardsolver.h>
#include <inputtools.h>
#include <mydofrenumbering.h>
#include <mypreconditioner.h>
#include <myvectortools.h>
#include <outputtools.h>
#include <myfe_nedelec.h>


using namespace dealii;
```

★La clase sphereBenchmark contiene un objeto de elemtnos finitos, una triangulación y un objeto que administra los grados de libertad. Ademaś, tiene un función que obtiene los valores de las contantes físicas del problema a partir de un archivo de entrada (void initialise_materials();), y una función que determina qué tipo de condiciones de frontera se utilizan y qué parámetros utilizar en cada celda(void process_mesh(bool neuman_flag);). La funcin void *std::string input_filename,std::string output_filename,const unsigned int href);* se encarga de generar la triangulación, ensamblar las matrices, solucionar el sistema lineal e imprimir lo datos en formato vtk.

```
namespace sphereBenchmark
{

  template <int dim>
  class sphereBenchmark
  {
  public:
```

```
    sphereBenchmark (const unsigned int poly_order,
                     const unsigned int mapping_order=2);
    ~sphereBenchmark ();
    void run(std::string input_filename,
             std::string output_filename,
             const unsigned int href);
  private:
    Triangulation<dim> tria; //Clase de dela.ii, triangulacion
    FESystem<dim> fe; //clase de deal.ii, objeto de elementos finitos
    DoFHandler<dim> dof_handler; //Clase de deal.ii, administrar los grados de libertad

    const unsigned int poly_order;
    const unsigned int mapping_order;

    void initialise_materials();
    void process_mesh(bool neuman_flag);
  };
```

★Estas funciones son el constructor y el destructor de un objeto perteneciente a la clase sphereBenchmark.

```
  //-------Constructor-------------------
  template <int dim>
  sphereBenchmark<dim>::sphereBenchmark(const unsigned int poly_order,
                                        const unsigned int mapping_order)
  :
  fe (MyFE_Nedelec<dim>(poly_order), mapping_order), //MyFE_Nedelec<> es una clase
  //definida en myfe_nedelec.h
  dof_handler (tria),
  poly_order(poly_order),
  mapping_order(mapping_order)
  {
  }
  //-----Destructor--------------
  template <int dim>
  sphereBenchmark<dim>::~sphereBenchmark ()
  {
    dof_handler.clear ();
  }
```

★ Las variables definidas bajo el namespace de EquationData guardan los valores de las contantes físicas del problema, somo lo son la permitividad de la esfera, la conductividad de la esfera, la frecuencia el campo magnético incidente, etc.

```
  //----initialise_materials--------------
  template <int dim>
  void sphereBenchmark<dim>::initialise_materials()
  {
    EquationData::param_mur.reinit(2);
    EquationData::param_mur(0) = EquationData::param_mur_background;
    EquationData::param_mur(1) = EquationData::param_mur_conducting;

    EquationData::param_sigma.reinit(2);
    EquationData::param_sigma(0) = EquationData::param_sigma_background;
    EquationData::param_sigma(1) = EquationData::param_sigma_conducting;

    EquationData::param_epsilon.reinit(2);
    EquationData::param_epsilon(0) = EquationData::param_epsilon_background;
    EquationData::param_epsilon(1) = EquationData::param_epsilon_conducting;

    EquationData::rhs_factor = 0.0; // No source term for this problem.
```

```
    // kappa = -omega^2*epr + i*omega*sigma;
    // i.e. kappa_re = -omega^2
    //      kappa_im = omega*sigma
    EquationData::param_kappa_re.reinit(EquationData::param_mur.size());
    EquationData::param_kappa_im.reinit(EquationData::param_mur.size());
    for (unsigned int i=0;i<EquationData::param_mur.size();i++) // note mur and kappa
    //must have same size:
    {
      // Unregularised - will be handled in forward solver.
      EquationData::param_kappa_re(i) = 0.0;
      EquationData::param_kappa_im(i) = EquationData::param_sigma(i)*EquationData::param_

    }
  }
```

★ Se asigna un identificador a las celdas que se encuentran en la frontera de la triangulación, para que el programa identifique las fronteras como esféricas(set_all_manifold_ids(100)). Además, se asignan condiciones de Neumann para la frontera (set_all_boundary_ids (10)) si neumann_flag=true, de lo contrario se asigna condiciones de Dirichlet (set_all_boundary_ids (0)).

```
  //------process_mesh-------------------------
  template <int dim>
  void sphereBenchmark<dim>::process_mesh(bool neumann_flag)
  {
    // Routine to process the read in mesh
    typename Triangulation<dim>::cell_iterator cell;
    const typename Triangulation<dim>::cell_iterator endc = tria.end();

    // Make the interior sphere's boundary a spherical boundary:
    // First set all manifold_ids to 0 (default).
    cell = tria.begin ();
    for (; cell!=endc; ++cell)
    {
      cell->set_all_manifold_ids(numbers::invalid_manifold_id);
      //numbers::invalid_manifold_id es un marcador de Deal.ii
    }
    // Now find those on the surface of the sphere.
    // Do this by looking through all cells with material_id
    // of the conductor, then finding any faces of those cells which
    // are shared with a cell with material_id of the non-conductor.
    cell = tria.begin ();
    for (; cell!=endc; ++cell)
    {
      // First if cell lies on boundary of the inner sphere,
      then flag as spherical (manifold 100).
      if (cell->material_id() == 1)
      {
        for (unsigned int face=0; face<GeometryInfo<dim>::faces_per_cell; ++face)
        {
          if (cell->neighbor(face)->material_id() == 0)
          {
            cell->face(face)->set_all_manifold_ids(100);
          }
        }
      }
      // Also if cell lies on outer boundary, then flag as spherical (manifold 100).
      else if (cell->at_boundary())
```

```
      {
        for (unsigned int face=0; face<GeometryInfo<dim>::faces_per_cell; ++face)
        {
          if (cell->face(face)->at_boundary())
          {
            cell->face(face)->set_all_manifold_ids(100);
          }
        }
      }
          if (cell->material_id() == 0)
      {
        cell-> set_all_manifold_ids(100);
      }
    }

    // Here we set the boundary ids for the boundary conditions
    // and may choose to mark the boundary as curved, etc.
    // Can also perform mesh refinement here.

    // Set boundaries to neumann (boundary_id = 10)
    if (neumann_flag)
    {
      cell = tria.begin ();
      for (; cell!=endc; ++cell)
      {
        for (unsigned int face=0; face<GeometryInfo<dim>::faces_per_cell; ++face)
        {
          if (cell->face(face)->at_boundary())
          {
            cell->face(face)->set_all_boundary_ids (10);
          }
        }
      }
    }
    else
    {
      cell = tria.begin ();
      for (; cell!=endc; ++cell)
      {
        for (unsigned int face=0; face<GeometryInfo<dim>::faces_per_cell; ++face)
        {
          if (cell->face(face)->at_boundary())
          {
            cell->face(face)->set_all_boundary_ids (0);
          }
        }
      }
    }
  }
```

★ Se crea la triangulación, se ensambla la matriz y se resuelve el sistema lineal. El archivo input_filename contiene los valores de las constantes físicas del problema, y el archivo output_filename contiene la solución del problema.
♣ El objeto InputTools::ParameterReader param(prm); se encarga de guardar los valores de las constantes físicas en las variables bajo el namespace de EquationData y MeshData.

```
//--------run--------------------------------
template <int dim>
void sphereBenchmark<dim>::run(std::string input_filename,
                               std::string output_filename,
```

```
                                const unsigned int href)
{

  ParameterHandler prm; //ParameterHandler clase perteneciente a deal.ii
  InputTools::ParameterReader param(prm); //Clase definida en InputTools
  param.read_and_copy_parameters(input_filename); //Leer los parametros del
  //archivo input_filename
```

♣ Si MeshData::external_mesh=true entonces la triagnulación se crea a partir de un archivo externo, de lo contrario, es creada dentro del programa, como se realiza en esta ocasión.

```
  const double sphere_radius=MeshData::inner_radius;
  static const SphericalManifold<dim> sph_boundary; //SphericalManifold clase
  //perteneciente a deal.ii
  if (!MeshData::external_mesh)//Si no hay un archivo con el mallado
  {

    const double ball_radius = sphere_radius; //sphere_radious=inner_radious
    // Make the central shell account for 1/4 of the shell part of the mesh:
    const double middle_radius = 0.25*(MeshData::radius - MeshData::inner_radius) +
    MeshData::inner_radius;
    const double outer_radius = MeshData::radius;
    Triangulation<dim> inner_ball;
    Triangulation<dim> outer_ball;
    Triangulation<dim> whole_ball;
    Triangulation<dim> middle_shell;
    Triangulation<dim> outer_shell;
```

♣ La triangulación utilizada corresponde al mallado de un cascarón esférico y está representada por la variable "tria". Es creada al fusionar la triangulación de dos cascarones esféricos concéntricos "middle_shellz .ºuter_shell". Se crea la frontera interna de la triangulación y se llama *whole_ball*, cuyo radio corresponde a la variable MeshData::inner_radious. Adiconalmente, aquellas celdas que tengan identificador 100 se les asigna una variedad esférica.

Además, se realiza un refinamiento de las celdas href veces al terminar de crear la triangulación.

```
    GridGenerator::hyper_ball(whole_ball,
                              Point<dim> (0.0,0.0,0.0),
                              sphere_radius); //Clase perteneciente a deal.ii, se
                              //crea la esfera interior (sphere_radious=inner_radious)
    typename Triangulation<dim>::cell_iterator cell = whole_ball.begin();
    typename Triangulation<dim>::cell_iterator endc = whole_ball.end();
    for (;cell!=endc; ++cell)
    {
      cell->set_material_id(1); //Tipo de material_id en la forntera interna,
      //debe ser conductor
    }

    GridGenerator::hyper_shell(middle_shell,
                               Point<dim> (0.0,0.0,0.0),
                               ball_radius,
                               middle_radius,
                               6); //Clase perteneciente a deal.ii, se crea el
                               //cascaron hasta middle_radious
                               //(ball_radious=inner_radious)
    cell = middle_shell.begin();
    endc = middle_shell.end();
    for (;cell!=endc; ++cell)
    {
      cell->set_material_id(0);//Tipo de material_id en el cascaron interno,
```

```
      //debe ser no conductor
    }
    Triangulation<dim> middle_ball;
    GridGenerator::merge_triangulations(whole_ball,
                                        middle_shell,
                                        middle_ball); //Clase perteneciente a deal.ii,
                                        //se crean la triangulacion middle_ball como
                                        //la union de while_ball y middle_shell

    GridGenerator::hyper_shell(outer_shell,
                               Point<dim> (0.0,0.0,0.0),
                               middle_radius,
                               outer_radius,
                               6); //Clase perteneciente a deal.ii,
                               //se crea el cascaron externo
    cell = outer_shell.begin();
    endc = outer_shell.end();
    for (;cell!=endc; ++cell)
    {
      cell->set_material_id(0);//Tipo de material_id en el cascaron externo,
      //debe ser no conductor
    }
    GridGenerator::merge_triangulations(middle_ball,
                                        outer_shell,
                                        tria); //Clase perteneciente a dela.ii, se
                                        //unen las triangulaciones midle_ball
                                        //y outer_ball para generar tria
    process_mesh(EquationData::neumann_flag); //Funcion interna que asigna condiciones
    //de frontera

    tria.set_manifold (100, sph_boundary);//Funcion de Triangulation de deal.ii,
    //asigna un manifold a una parte de la triangulacion;
    //asigna un manifold esferico a todas las lartes que estan marcadas con 100

    if (href>0)
    {
      tria.refine_global(href); //Funcion de Triangulation de deal.ii,
      //refina href veces el mallado de la triangulacion tria
    }

  }
```

♣ Si existe un archivo con la información de la triangulación se toman los datos de este y se refina href veces.

```
  else //Si hay un archivo con el mallado
  {
    InputTools::read_in_mesh<dim>(IO_Data::mesh_filename,
                                  tria); //Leer el mallado del archivo externo



    process_mesh(EquationData::neumann_flag); //Funcion interna que asigna
    //condiciones de frontera
    // Set the marked boundary to be spherical:
    tria.set_manifold (100, sph_boundary); //Funcion de Triangulation de deal.ii,
    //asigna una variedad esferica a las celdas cuyo id es 100
    if (href>0)
    {
      tria.refine_global(href); //Funcion de Triangulation de deal.ii,
```

```
        //refina href veces el mallado de la triangulacion tria
      }
    }
```

♣ Se cargan los valores de las constantes físicas a partir del archivo input_filename.

```
initialise_materials(); //funcion de la clase sphereBenchmark
```

♣ Se ensambla la matriz y se inicia el solucionador del problema. La clase que contiene las rutinas para ensamblar y solucionar el sistema lineal se llama ForwardSolver::EddyCurrent y está definida en los archivos "forwardsolver.cz "forwardsolver.cc". Después de ensamblar la matriz se inicia el pre-condicionador, el cual se utiliza para resolver el problema lineal rápidamente.

```
deallog << "Number␣of␣active␣cells:␣␣␣␣␣␣␣"
<< tria.n_active_cells() //Funcion de Triangulation de dela.ii. Cantidad total
//de celdas activas en la triangualcion tria
<< std::endl;

// Now setup the forward problem:
dof_handler.distribute_dofs (fe); //Clase perteneciente a deal.ii,
//.distribute.dofs(fe) indexa los grados de
//libertad del objeto de elementos finitos fe

const MappingQ<dim> mapping(mapping_order, (mapping_order>1 ? true : false));
//Clase de deal.ii. Se utilizan polinomios de orden mapping_order en todas las
//celdas; si mapping_order >1 entonces se utilizan polinomios de orden
//mapping_order, en caso contrario se utilizan las bases de menos orden
//posbile segun la triangulacion y las funciones base.

//const MappingQ1<dim> mapping;
ForwardSolver::EddyCurrent<dim, DoFHandler<dim>> eddy(mapping,
                                                      dof_handler,
                                                      fe,
 PreconditionerData::use_direct); //Clase definida por el usuario

deallog << "Number␣of␣degrees␣of␣freedom:␣"
<< dof_handler.n_dofs() //funcion de DoFHandler de dela.ii.
//Cantidad de grados de libertad.
<< std::endl;

// assemble the matrix for the eddy current problem:
deallog << "Assembling␣System␣Matrix...." << std::endl;
eddy.assemble_matrices(dof_handler); //Funcion de la clase EddyCurrent
//definida por el ususario
deallog << "Matrix␣Assembly␣complete.␣" << std::endl;

// initialise the linear solver - precomputes any inverses
//for the preconditioner, etc:
deallog << "Initialising␣Solver..." << std::endl;
eddy.initialise_solver(); //Funcion de la clase EddyCurrent definida
//por el ususario
deallog << "Solver␣initialisation␣complete.␣" << std::endl;
```

♣ Se utiliza la información sobre el campo incidente para calcular el vector de respuesta, el cual es creado con la función .[ed]dy.assemble_rhs(dof_handler,*boundary_conditions);". Se puede tener información sobre el vector de polarización del campo incidente o sobre la magnitud del campo magnético incidente.

```
// construct RHS for this field:
Vector<double> uniform_field(dim+dim); //La clase Vector<double> pertenece a deal.ii
for (unsigned int d=0;d<dim;++d)
{
```

```
  uniform_field(d) = PolarizationTensor::H0[0](d);//Parte real del campo incidente
}

EddyCurrentFunction<dim>* boundary_conditions; //Se crea un funcion de deal.ii que
//calcula las variables relacionadas con el campo incidente
if (PolarizationTensor::enable)
{
  boundary_conditions
  = new backgroundField::conductingObject_polarization_tensor<dim>
      (PolarizationTensor::H0,
       PolarizationTensor::polarizationTensor); //usar condiciones de frontera
       //para un vector de polarizacion conocido
}
else
{
  boundary_conditions
  = new backgroundField::conductingSphere<dim> (sphere_radius,
  PolarizationTensor::H0); //usar condiciones de frontera
  //para un campo magnetico incidente conocido
}

// assemble rhs
deallog << "Assembling␣System␣RHS...." << std::endl;
eddy.assemble_rhs(dof_handler,
                  *boundary_conditions); //funcion de la clase
                  //ForwardSolver::EddyCurrent<dim, DoFHandler<dim>>
deallog << "Matrix␣RHS␣complete.␣" << std::endl;
```

♣ Se soluciona el sistema lineal y se calcula el error cometido respecto de la solución teórica. La solución se guarda en el vector "solutionz se utiliza para comparar la con la solución teórica; en el archivo "myvectortools.cc"se define el valor de la solución teórica y se implementa un algoritmo que la compara con la solución numérica; función "MyVectorTools::calcErrorMeasures()".

```
deallog << "Solving...␣" << std::endl;
Vector<double> solution; //La clase Vector<double> pertenece a deal.ii
// solve system & storage in the vector of solutions:
unsigned int n_gmres_iterations;
eddy.solve(solution,n_gmres_iterations); //funcion de la clase
//ForwardSolver::EddyCurrent<dim, DoFHandler<dim>>, n_gmres_iterations de termina
//la cantidad de iteraciones usadas al solucionar el problema

deallog << "Computed␣solution.␣" << std::endl;

// Output error to screen:

double l2err_wholeDomain;
double l2err_conductor;
double hcurlerr_wholeDomain;
double hcurlerr_conductor;

MyVectorTools::calcErrorMeasures(mapping,
                                 dof_handler,
                                 solution,
                                 *boundary_conditions,
                                 l2err_wholeDomain,
                                 l2err_conductor,
                                 hcurlerr_wholeDomain,
                                 hcurlerr_conductor);
```

```
deallog << "L2␣Errors␣|␣Whole␣Domain:␣" << l2err_wholeDomain << "␣␣␣Conductor␣Only:"
<< l2err_conductor << std::endl;
deallog << "HCurl␣Error␣|␣Whole␣Domain:␣" << hcurlerr_wholeDomain << "␣␣␣Conductor␣On
<< hcurlerr_conductor << std::endl;
// Short version:
std::cout << tria.n_active_cells()
<< "␣" << dof_handler.n_dofs()
<< "␣" << n_gmres_iterations
<< "␣" << l2err_wholeDomain
<< "␣" << l2err_conductor
<< "␣" << hcurlerr_wholeDomain
<< "␣" << hcurlerr_conductor
<< std::endl;
```

♣ Se crean los archivos que contienen los datos que se desean analizar y/o visualizar. Ya que el código está diseñado para los tres problemas expuesto del estándar TEAM, entonces, se tienen en cuenta si el mallado es un cubo, un cilindro o una esfera para generar 4 archivos adicionales, los cuales contienen el valor del campo eléctrico sobre 4 ejes diferentes. Al terminar de ejecutar la función "sphereBenchmark<dim>::run()"se borra el puntero que contiene la función utilizada para calcular las condiciones de frontera, esto evita problemas de administración de memoria caché.

```
{
  std::ostringstream tmp;
  tmp << output_filename;
  OutputTools::output_to_vtk<dim, DoFHandler<dim>>(mapping,
                                                   dof_handler,
                                                   solution,
                                                   tmp.str(),
                                                   *boundary_conditions);
}

// Output the solution fields along a line to a text file:
Point <dim> xaxis;
Point <dim> yaxis;
Point <dim> zaxis;
Point <dim> diagaxis;
if (MeshData::boundary_shape=="cube")
{
  xaxis = Point<dim> (MeshData::xmax, 0.0, 0.0);
  yaxis = Point<dim> (0.0, MeshData::ymax, 0.0);
  zaxis = Point<dim> (0.0, 0.0, MeshData::zmax);
  diagaxis = Point<dim> (MeshData::xmax, MeshData::ymax, MeshData::zmax);
}
else if (MeshData::boundary_shape=="cylinder_z")
{
  xaxis = Point<dim> (MeshData::radius, 0.0, 0.0);
  yaxis = Point<dim> (0.0, MeshData::radius, 0.0);
  zaxis = Point<dim> (0.0, 0.0, MeshData::zmax);
  diagaxis = Point<dim> (MeshData::radius*cos(numbers::PI/4.0),
  MeshData::radius*sin(numbers::PI/4.0), MeshData::zmax);
}
else if (MeshData::boundary_shape=="sphere")
{
  xaxis = Point<dim> (MeshData::radius, 0.0, 0.0);
  yaxis = Point<dim> (0.0, MeshData::radius, 0.0);
  zaxis = Point<dim> (0.0, 0.0, MeshData::radius);
  diagaxis = Point<dim> (MeshData::radius/sqrt(3),
  MeshData::radius/sqrt(3), MeshData::radius/sqrt(3));
```

```
    }

    {
    std::stringstream tmp;
    tmp << output_filename << "_xaxis";
    std::string xaxis_str = tmp.str();
    OutputTools::output_radial_values<dim> (mapping,
                                            dof_handler,
                                            solution,
                                            *boundary_conditions,
                                            uniform_field,
                                            xaxis,
                                            xaxis_str);
    }
    {
      std::stringstream tmp;
      tmp << output_filename << "_yaxis";
      std::string yaxis_str = tmp.str();
      OutputTools::output_radial_values<dim> (mapping,
                                              dof_handler,
                                              solution,
                                              *boundary_conditions,
                                              uniform_field,
                                              yaxis,
                                              yaxis_str);
    }
    {
      std::stringstream tmp;
      tmp << output_filename << "_zaxis";
      std::string zaxis_str = tmp.str();
      OutputTools::output_radial_values<dim> (mapping,
                                              dof_handler,
                                              solution,
                                              *boundary_conditions,
                                              uniform_field,
                                              zaxis,
                                              zaxis_str);
    }
    {
      std::stringstream tmp;
      tmp << output_filename << "_diagaxis";
      std::string diagaxis_str = tmp.str();
      OutputTools::output_radial_values<dim> (mapping,
                                              dof_handler,
                                              solution,
                                              *boundary_conditions,
                                              uniform_field,
                                              diagaxis,
                                              diagaxis_str);
    }

    delete boundary_conditions;
  }
  //________sphereBenchmark,end___________________________
}
//*******sphereBenchmark,end*******************************
```

★ La parte principal del programa implementa un ciclo para que el usuario puede ingresar el valor de "p_order", "href", "mapping_order", .ºutput_filamez ïnput_filename". Luego, se crea un ob-

jeto de la clase "sphereBenchmark::sphereBenchmark<dim>"llamado .edy_voltagesz se ejecuta la función .edy_voltages.run(input_filename,output_filename,href);"la cual soluciona el problema de elementos finitos y genera los archivos con los datos.

```
int main (int argc, char* argv[])
{
  using namespace dealii;

  const int dim = 3;
  // Set default input:
  unsigned int p_order = 0;
  unsigned int href = 0;
  unsigned int mapping_order = 1;
  std::string output_filename = "sphere";
  std::string input_filename = "../input_files/sphere_benchmark.prm";

  // Allow for input from command line:
  if (argc > 0)
  {
    for (int i=1;i<argc;i++)
    {
      if (i+1 != argc)
      {
        std::string input = argv[i];
        if (input == "-p")
        {
          std::stringstream strValue;
          strValue << argv[i+1];
          strValue >> p_order;
        }
        if (input == "-m")
        {
          std::stringstream strValue;
          strValue << argv[i+1];
          strValue >> mapping_order;
          if (mapping_order < 1)
          {
            std::cout << "ERROR:␣mapping␣order␣must␣be␣>␣0" << std::endl;
            return 1;
          }
        }
        if (input == "-i")
        {
          input_filename = argv[i+1];
        }
        if (input == "-o")
        {
          output_filename = argv[i+1];
        }
        if (input == "-h") // h refinement
        {
          std::stringstream strValue;
          strValue << argv[i+1];
          strValue >> href;
        }
      }
    }
  }
```

```
  // Only output to logfile, not console:
  deallog.depth_console(0);
  std::ostringstream deallog_filename;
  deallog_filename << output_filename << "_p" << p_order << ".deallog";
  std::ofstream deallog_file(deallog_filename.str());
  deallog.attach(deallog_file);

  sphereBenchmark::sphereBenchmark<dim> eddy_voltages(p_order,
                                                      mapping_order); //Clase definida
                                                      //en este archivo
  eddy_voltages.run(input_filename,
                    output_filename,
                    href); //Funcion de la clase spehreBenchmark<dim>

  deallog_file.close();
  return 0;
}
```

B. forwardsolver.cc

★ La función "void EddyCurrent<dim, DH>::assemble_matrices (const DH &dof_handler)"se encarga de ensamblar la matriz de masa y la matriz del rotacional; para ello utiliza una fórmula de cuadratura de orden "quad_order". Ya que el sistema de compone de una parte real y una parte imaginaria entonces la matriz se divide en cuatro bloques de la misma dimensión:

$$\hat{K}x = (\hat{A} + \hat{M})\overrightarrow{x} = \overrightarrow{b}$$

$$\Rightarrow \begin{pmatrix} A & M_i \\ -M - i & -A \end{pmatrix} \begin{pmatrix} x_r \\ x_i \end{pmatrix} = \begin{pmatrix} b_r \\ b_i \end{pmatrix}$$

De esta forma el sistema lineal es simétrico y real. En este caso $\hat{A}$ s la matriz del rotacional, y M la matriz de masa, y por la definición de las funciones base las matriz local del rotacional es diagonal.

En esta función se calculan las matrices locales de los grados de libretad no restringidos, y se se agregan al precondicionardor.

```
 template <int dim, class DH>
  void EddyCurrent<dim, DH>::assemble_matrices (const DH &dof_handler)
  {
    /*
     * Function to assemble the system matrix.
     *
     * Should really only need to be called once for a given
     * set of material parameters.
     */
    QGauss<dim>  quadrature_formula(quad_order);

    const unsigned int n_q_points = quadrature_formula.size();

    const unsigned int dofs_per_cell = fe->dofs_per_cell;

    FEValues<dim> fe_values (*mapping,
                             *fe, quadrature_formula,
                             update_values    |  update_gradients |
                             update_quadrature_points  |  update_JxW_values);
```

```
// Extractors to real and imaginary parts
const FEValuesExtractors::Vector E_re(0);
const FEValuesExtractors::Vector E_im(dim);

std::vector<FEValuesExtractors::Vector> vec(2);
vec[0] = E_re;
vec[1] = E_im;

// Local cell storage:
FullMatrix<double> cell_matrix (dofs_per_cell, dofs_per_cell);
FullMatrix<double> cell_preconditioner (dofs_per_cell, dofs_per_cell);
std::vector<types::global_dof_index> local_dof_indices (dofs_per_cell);

// Setup regularisation
const double kappa_regularised[2][2]
= { {0.0, -EquationData::param_regularisation},
    {-EquationData::param_regularisation, 0.0} };

const typename DH::active_cell_iterator endc = dof_handler.end();
typename DH::active_cell_iterator cell;

cell = dof_handler.begin_active();

for (; cell!=endc; ++cell)
{
  fe_values.reinit (cell);
  cell->get_dof_indices (local_dof_indices);

  const double current_mur = EquationData::param_mur(cell->material_id());
  const double mur_inv = 1.0/current_mur;
  const double current_kappa_re = EquationData::param_kappa_re(cell->material_id());
  const double current_kappa_im = EquationData::param_kappa_im(cell->material_id());
  // for preconditioner:
  const double current_kappa_magnitude = sqrt(current_kappa_im*current_kappa_im + cur

  // Store coefficients of the real/imaginary blocks:
  // NOTE, to make matrix symmetric, we multiply the bottom row of blocks by -1
  const double mur_matrix[2][2] = {
    {mur_inv, 0.0},
    {0.0, -mur_inv} };

  const double kappa_matrix[2][2] = {
    {current_kappa_re, -current_kappa_im},
    {-current_kappa_im, -current_kappa_re} };

  const double kappa_matrix_precon[2][2] = {
    {current_kappa_magnitude, 0.0},
    {0.0, current_kappa_magnitude} };

  cell_matrix = 0;
  cell_preconditioner = 0;
  for (unsigned int i=0; i<dofs_per_cell; ++i)
  {
    if (!constraints.is_constrained(local_dof_indices[i]))
    {
      const unsigned int block_index_i = fe->system_to_block_index(i).first;
```

```
// Construct local matrix:
for (unsigned int j=i; j<dofs_per_cell; ++j)
{
  if (!constraints.is_constrained(local_dof_indices[j]))
  {
    const unsigned int block_index_j = fe->system_to_block_index(j).first;
    double mass_part = 0;
    double curl_part = 0;

    const double mu_term = mur_matrix[block_index_i][block_index_j];
    const double precon_kappa_term = kappa_matrix_precon[block_index_i][block_i
    double kappa_term;
    // Are we in the conductor?
    // If no, then check if we're in the lowest order block
    // if yes, then use regularised kappa.
    if (cell->material_id() != 1
      && local_dof_indices[i] < end_lowest_order_dofs
      && local_dof_indices[j] < end_lowest_order_dofs)
    {
      kappa_term = kappa_regularised[block_index_i][block_index_j];
    }
    // Otherwise, either in the conductor or outside of the lowest order block
    // So just use the normal kappa value.
    else
    {
      kappa_term = kappa_matrix[block_index_i][block_index_j];
    }
    if (block_index_i == block_index_j)
    {
      for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
      {
        curl_part
        += fe_values[vec[block_index_i]].curl(i, q_point)
        *fe_values[vec[block_index_j]].curl(j, q_point)
        *fe_values.JxW(q_point);

        mass_part
        += fe_values[vec[block_index_i]].value(i, q_point)
        *fe_values[vec[block_index_j]].value(j, q_point)
        *fe_values.JxW(q_point);
      }

      // Use skew-symmetry to fill matrices:
      cell_matrix(i,j) = mu_term*curl_part + kappa_term*mass_part;
      cell_matrix(j,i) = cell_matrix(i,j);

      if (!direct)
      {
        // Lowest order block, matches cell matrix
        // This is a single block, containing real and imaginary.
        if (local_dof_indices[i] < end_lowest_order_dofs
          && local_dof_indices[j] < end_lowest_order_dofs)
        {
          cell_preconditioner(i,j) = cell_matrix(i,j);
          cell_preconditioner(j,i) = cell_matrix(j,i);
        }
        // Real & imaginary higher order gradient blocks, M(|k|)
        else if ( (local_dof_indices[i] >= end_lowest_order_dofs
```

```
                  && local_dof_indices[j] >= end_lowest_order_dofs
                  && local_dof_indices[i] < end_gradient_dofs_real
                  && local_dof_indices[j] < end_gradient_dofs_real)
                  || (local_dof_indices[i] >= end_nongradient_dofs_real
                  && local_dof_indices[j] >= end_nongradient_dofs_real
                  && local_dof_indices[i] < end_gradient_dofs_imag
                  && local_dof_indices[j] < end_gradient_dofs_imag) )
                {
                  cell_preconditioner(i,j) = precon_kappa_term*mass_part;
                  cell_preconditioner(j,i) = cell_preconditioner(i,j);
                }
                // Real & imaginary higher order non-gradient block, C+M(|k|)
                // Note we keep coeff of C in the precon positive.
                else
                {
                  cell_preconditioner(i,j) = mur_inv*curl_part + precon_kappa_term*mass
                  cell_preconditioner(j,i) = cell_preconditioner(i,j);
                }
              }
            }
            else // off diagonal - curl-curl operator not part of these blocks..
            {
              for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
              {
                mass_part
                += fe_values[vec[block_index_i]].value(i,q_point)
                *fe_values[vec[block_index_j]].value(j,q_point)
                *fe_values.JxW(q_point);
              }
              // Use skew-symmetry to fill matrices:
              cell_matrix(i,j) = kappa_term*mass_part;
              cell_matrix(j,i) = cell_matrix(i,j);

              // Only need  to fill preconditoner for the lowest order block
              // for the off-diagonals.
              if (!direct
                && local_dof_indices[i] < end_lowest_order_dofs
                && local_dof_indices[j] < end_lowest_order_dofs)
              {
                cell_preconditioner(i,j) = cell_matrix(i,j);
                cell_preconditioner(j,i) = cell_matrix(j,i);
              }
            }
          }
        }
      }
    }
    // Distribute & constraint to global matrices:
    // Note: need to apply RHS constraints using the columns of their
    // local matrix in the other routine.
    constraints.distribute_local_to_global(cell_matrix,
                                           local_dof_indices,
                                           system_matrix);
    if (!direct)
    {
      constraints.distribute_local_to_global(cell_preconditioner,
                                             local_dof_indices,
                                             system_preconditioner);
```

```
      }
    }
  }
```

★ Esta función calcula el vector constante asociado al sistema lineal y las matrices locales asociados a grados de libertad restringidos:

1. Grados de libertad restringidos: En la región que no tiene conductor se impone que el gradiente de la inducción eléctrica sea nulo, i.e. no hay acumulación de carga fuera del conductor.

2. Condiciones de Neumann: En la frontera externa (la esfera de mayor radio), se conoce el valor del campo magnético, por lo tanto, se tiene que $H|_{\partial\Omega} = curl(A)|_{\partial\Omega}$, lo cual añade una restricción a las funciones base sobre la frontera.

```
// Assemble the rhs for a zero RHS
// in the governing equation.
template <int dim, class DH>
void EddyCurrent<dim, DH>::assemble_rhs (const DH &dof_handler,
  const EddyCurrentFunction<dim> &boundary_function)
{
  // Function to assemble the RHS for a given boundary_function,
  // which implements the RHS of the equation and
  // Dirichlet & Neumann conditions
  //
  // It first updates the constraints and then computes the RHS.
  //
  // NOTE: this will completely reset the RHS stored within the class.

  // Zero the RHS:
  system_rhs = 0;

  compute_constraints(dof_handler,
                      boundary_function);

  QGauss<dim>  quadrature_formula(quad_order);
  const unsigned int n_q_points = quadrature_formula.size();

  QGauss<dim-1> face_quadrature_formula(quad_order);
  const unsigned int n_face_q_points = face_quadrature_formula.size();

  const unsigned int dofs_per_cell = fe->dofs_per_cell;

  // Needed to calc the local matrix for distribute_local_to_global
  // Note: only need the columns of the constrained entries.
  FEValues<dim> fe_values (*mapping,
                           *fe, quadrature_formula,
                           update_values    |  update_gradients |
                           update_quadrature_points  |  update_JxW_values);

  FEFaceValues<dim> fe_face_values(*mapping,
                                   *fe, face_quadrature_formula,
                                   update_values | update_quadrature_points |
                                   update_normal_vectors | update_JxW_values);

  // Extractors to real and imaginary parts
  const FEValuesExtractors::Vector E_re(0);
  const FEValuesExtractors::Vector E_im(dim);
```

```
    std::vector<FEValuesExtractors::Vector> vec(2);
    vec[0] = E_re;
    vec[1] = E_im;

    // Local cell storage:
    Vector<double> cell_rhs (dofs_per_cell);
    std::vector<types::global_dof_index> local_dof_indices (dofs_per_cell);
    FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);

    // Setup regularisation
    const double kappa_regularised[2][2]
    = { {0.0, -EquationData::param_regularisation},
        {-EquationData::param_regularisation, 0.0} };

    typename DoFHandler<dim>::active_cell_iterator
    cell = dof_handler.begin_active(),
    endc = dof_handler.end();
    for (; cell!=endc; ++cell)
    {
      cell_rhs = 0.0;
      // Material parameters:
      const double current_mur = EquationData::param_mur(cell->material_id());
      const double mur_inv = 1.0/current_mur;
      const double current_kappa_re = EquationData::param_kappa_re(cell->material_id());
      const double current_kappa_im = EquationData::param_kappa_im(cell->material_id());
      // Store coefficients of the real/imaginary blocks:
      // NOTE, to make matrix symmetric, we multiply the bottom row of blocks by -1
      // RHS coeff:
      const double rhs_coefficient[2] = {
        EquationData::rhs_factor,
        -EquationData::rhs_factor};
      // mu_r coeff (note it's real valued, so zero on off-diag).
      const double mur_matrix[2][2] = {
        {mur_inv, 0.0},
        {0.0, -mur_inv} };
      // kappa coeff
      const double kappa_matrix[2][2] = {
        {current_kappa_re, -current_kappa_im},
        {-current_kappa_im, -current_kappa_re} };

      // Update the fe values object.
      fe_values.reinit (cell);
      cell->get_dof_indices (local_dof_indices);

      // Loop for non-zero right hand side in equation:
      // Only needed if the RHS is non-zero.
      if (!boundary_function.zero_rhs())
        {
          // RHS Storage:
          std::vector<Vector<double> > rhs_value_list(n_q_points, Vector<double>(
      fe->n_components()));
          Tensor<1,dim> rhs_value;

          boundary_function.rhs_value_list(fe_values.get_quadrature_points(),
      rhs_value_list, cell->material_id());
          for (unsigned int i=0; i<dofs_per_cell; ++i)
            {
              if (!constraints.is_constrained(local_dof_indices[i]))
```

```
                {
                  const unsigned int block_index_i = fe->system_to_block_index(i).first;
                  const unsigned int d_shift = block_index_i*dim;
                  double rhs_term = 0;

                  for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
                    {
                      for (unsigned int d=0; d<dim; ++d)
                        {
                          rhs_value[d] = rhs_value_list[q_point](d+d_shift);
                        }
                      rhs_term += rhs_value
                        *fe_values[vec[block_index_i]].value(i,q_point)
                        *fe_values.JxW(q_point);
                    }
                  // Remember the J_{s} term is multiplied by mu0, this
                  // is communicated by EquationData::rhs_factor, which
                  // has been passed into rhs_coefficient, above.
                  cell_rhs(i) = rhs_coefficient[block_index_i]*rhs_term;
                }
            }
        }

      // Loop over faces for neumann condition:
      // Only needed if the neumann values are non-zero.
      if (!boundary_function.zero_curl())
      {
        // Neumann storage
        std::vector< Vector<double> > neumann_value_list(n_face_q_points,
        Vector<double>(fe->n_components()));
        Tensor<1,dim> normal_vector;
        Tensor<1,dim> neumann_value;
        Tensor<1,dim> crossproduct_result;

        for (unsigned int face_number=0; face_number<GeometryInfo<dim>::faces_per_cell;
        ++face_number)
        {
          if (cell->face(face_number)->at_boundary()
            &&
            (cell->face(face_number)->boundary_id() == 10))
          {
            fe_face_values.reinit (cell, face_number);

            boundary_function.curl_value_list(fe_face_values.get_quadrature_points(),
            neumann_value_list);
            // new
            for (unsigned int i=0; i<dofs_per_cell; ++i)
            {
              if (!constraints.is_constrained(local_dof_indices[i]))
              {
                const unsigned int block_index_i = fe->system_to_block_index(i).first;
                const unsigned int d_shift = block_index_i*dim;
                double neumann_term = 0;

                const double mu_term = mur_matrix[block_index_i][block_index_i];

                for (unsigned int q_point=0; q_point<n_face_q_points; ++q_point)
                {
```

```
                for (unsigned int d=0; d<dim; ++d)
                {
                  neumann_value[d] = neumann_value_list[q_point](d+d_shift);
                  normal_vector[d] = fe_face_values.normal_vector(q_point)(d);
                }
                cross_product(crossproduct_result, normal_vector, neumann_value);

                neumann_term -= crossproduct_result
                *fe_face_values[vec[block_index_i]].value(i,q_point)
                *fe_face_values.JxW(q_point);
              }
              // Note: mur_matrix factors in the sign of the block.
              cell_rhs(i) += mu_term*neumann_term;
            }
          }
        }
      }
    }

    // Create the columns of the local matrix which belong to a constrained
    // DoF. These are all that are needed to apply the constraints to the RHS
    cell_matrix = 0;

    for (unsigned int j=0; j<dofs_per_cell; ++j)
    {
      // Check each column to see if it corresponds to a constrained DoF:

      if ( constraints.is_inhomogeneously_constrained(local_dof_indices[j]) )
      {
        const unsigned int block_index_j = fe->system_to_block_index(j).first;
        // If yes, cycle through all rows to fill the column:
        for (unsigned int i=0; i<dofs_per_cell; ++i)
        {
          const unsigned int block_index_i = fe->system_to_block_index(i).first;

          double curl_part = 0;
          double mass_part = 0;

          const double mu_term = mur_matrix[block_index_i][block_index_j];
          double kappa_term;
          // Are we in the conductor?
          // If no, then check if we're in the lowest order block
          // if yes, then use regularised kappa.
          if (cell->material_id() != 1
            && local_dof_indices[i] < end_lowest_order_dofs
            && local_dof_indices[j] < end_lowest_order_dofs)
          {
            kappa_term = kappa_regularised[block_index_i][block_index_j];
          }
          // Otherwise, either in the conductor or outside of the lowest order block
          // So just use the normal kappa value.
          else
          {
            kappa_term = kappa_matrix[block_index_i][block_index_j];
          }
          if (block_index_i == block_index_j)
          {
            for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
```

```
              {
                curl_part
                += fe_values[vec[block_index_i]].curl(i, q_point)
                *fe_values[vec[block_index_j]].curl(j, q_point)
                *fe_values.JxW(q_point);

                mass_part
                += fe_values[vec[block_index_i]].value(i, q_point)
                *fe_values[vec[block_index_j]].value(j, q_point)
                *fe_values.JxW(q_point);
              }
              cell_matrix(i,j) = mu_term*curl_part + kappa_term*mass_part;
            }
            else // off diagonal - curl-curl operator not needed.
            {
              for (unsigned int q_point=0; q_point<n_q_points; ++q_point)
              {
                mass_part
                += fe_values[vec[block_index_i]].value(i,q_point)
                *fe_values[vec[block_index_j]].value(j,q_point)
                *fe_values.JxW(q_point);
              }
              cell_matrix(i,j) = kappa_term*mass_part;
            }
          }
        }
      }
      // Use the cell matrix constructed for the constrained DoF columns
      // to add the local contribution to the global RHS:
      constraints.distribute_local_to_global(cell_rhs, local_dof_indices, system_rhs,
      cell_matrix);
    }
  }
```

★ El archivo "forwardsolver.ccçontiene más funciones, las cuales se encargan de crear un interfaz entre los elementos finitos definidos implementados por R.M. Kynch e implementar un pro-condicionador para solucionar el sistema lineal.