Preconditioners based on splitting for time domain photonic crystal modeling by matrix exponential Krylov subspace methods

Jeroen de Cloet
Jasper Marissen
Ties Westendorp

Tutor: Dr. M.A. Botchev

June 22, 2015

Abstract

Today the Maxwell equations of electrodynamics are still solved using the finite-difference time-domain method, because of the relative simplicity of the method. In order to solve these equations on an infinite domain, absorbing boundary conditions need to be imposed. We consider Maxwell's equations discretized in space only (using the Yee scheme) with a PML on the boundary to absorb outgoing waves. To solve the resulting system of ODEs we use exponential time integration. Because we solve the Maxwell equations in 3D, direct sparse linear solvers are not efficient for these systems. Therefore we have to use preconditioned iterative solvers based on the Krylov subspace methods. These methods require a solution of a linear system with shifted Maxwell operator. In this work we compare the accuracy and speed (in terms of computation time) of different splitting-based predontitioners of the discretized Maxwell operator for several interesting test cases.

Keywords: Maxwell equations, photonic crystal modeling, finite-difference time-domain, exponential time integration, Krylov SAI, preconditioning, splitting.

Contents

1 Introduction

The behavior of several interesting structures, such as photonic crystals with point defects or wave guides (linear defects), can be analyzed by solving the Maxwell equations of electrodynamics.

First we will state the Maxwell equations without source terms and derive a perfectly matched layer in three dimensions. We will do a numerical check to see if the perfectly matched layer works using an existing solver. Next we propose another way to solve the Maxwell equations numerically using the Krylov Shift-And-Invert method and generalized minimal residual method (GMRES). GMRES can be used with of without preconditioning. We look at three different cases: no preconditioning, ADI-splitting and field-splitting. Also we use BiCGSTAB instead of GMRES for the same three cases. For each different case we check the computational time and compare this with the existing solver.

Finally we look at how a Gaussian pulse propagates through a photonic crystal with a line defect. In one case the line defect has no boundaries. We expect that the waves decay inside the perfectly matched layer and therefore decay completely. In the other case the line defect has atoms on both ends. Here we expect that the most of the waves stay between these atoms and some of the waves propagate through the crystal and enter the perfectly matched layer.

2 Maxwell equations

At the core of any physical theory, there is typically some mathematical model upon which all analysis is done in order to derive meaningful results. For electrodynamics this model comes in the form of four differential equations collectively known as the Maxwell equations (or Maxwell's equations). They describe how the electric vector field $\mathbf{E} = [E_x(x,y,z,t)\ E_y(x,y,z,t)\ E_z(x,y,z,t)]^T$ and the magnetic vector field $\mathbf{H} = [H_x(x,y,z,t)\ H_y(x,y,z,t)\ H_z(x,y,z,t)]^T$ change with time. In fact they tell us that both fields propagate as waves and that in vacuum these waves travel at the speed of light, implying that the waves themselves must be light waves [1,2]. The Maxwell equations without source terms are given below:

Maxwell equations

$$\begin{aligned}
\nabla \cdot \mathbf{D} &= 0, && \text{(Gauss's law)}\\
\nabla \cdot \mathbf{B} &= 0, && \text{(Gauss's law for magnetism)}\\
\nabla \times \mathbf{E} &= -\mu \partial_t \mathbf{H}, && \text{(Maxwell-Faraday equation)}\\
\nabla \times \mathbf{H} &= \quad \epsilon \partial_t \mathbf{E}, && \text{(Ampère's circuital law)}
\end{aligned}$$

where we have the electric flux density $\mathbf{D} = \epsilon \mathbf{E}$ and the magnetic flux density $\mathbf{B} = \mu \mathbf{H}$.

Gauss's law states that the divergence of the electric flux density $\mathbf{D}$ is constant. This means that there can be some stationary source or sink and in case there is no charge the divergence is zero. Gauss's law of magnetism says that the same does not apply to the magnetic flux density $\mathbf{B}$; the divergence of the magnetic flux density is zero everywhere, implying that there is no beginning or end. In other words: magnetic monopoles (magnetic sources or sinks) do not exist. The Maxwell-Faraday equation tells us that a changing magnetic field $\mathbf{H}$ induces an electrical current. Finally, Ampère's circuital law states that a change in the electric field $\mathbf{E}$ produces a magnetic current.

2.1 Solving the Maxwell equations

It can easily be shown that if the initial conditions for $\mathbf{H}$ and $\mathbf{E}$ satisfy Gauss's law and Gauss's law for magnetism, then the solution will satisfy them for all time [3]. Hence we are only interested in solving the Maxwell-Faraday equation and Ampère's circuital law. These two equations can be rewritten to:

$$\partial_t \mathbf{H} = -\mu^{-1}(\nabla \times \mathbf{E}), \tag{1}$$

$$\partial_t \mathbf{E} = \quad \epsilon^{-1}\,(\nabla \times \mathbf{H}). \tag{2}$$

These equations can be brought to a compact form (with $\mathbf{y} = [H_x\ H_y\ H_z\ E_x\ E_y\ E_z]^T$):

$$\mathbf{y}'(t) = A\mathbf{y}(t) = \begin{pmatrix} 0 & -\frac{1}{\mu}\nabla\times \\ \frac{1}{\epsilon}\nabla\times & 0 \end{pmatrix} \mathbf{y}(t), \tag{3}$$

where $\mu = \mu(x,y,z)$ and $\epsilon = \epsilon(x,y,z)$.

In Section 3, when we have discretized the curl in the Maxwell operator A, this system of PDEs will be reduced to a system of ODEs with general solution: $\mathbf{y}(t) = e^{At}\mathbf{y}(0)$. If we have some way of approximating the matrix exponential (or its action), then we have a way to approximate solutions to our system of PDEs (as an alternative to discretizing both space and time).

2.2 Perfectly matched layer

Since our purpose is to simulate electromagnetic waves in an unbounded domain, we need to find a way to bound our domain without influencing the solution within some region of interest. To that end we will add a perfectly matched layer (PML) absorbing boundary [4]. In his initial paper, Bérenger derived the so-called *split-field* PML. Better techniques are used now, such as the *stretched coordinate* PML [2, 5, 6]. When performing a complex coordinate stretching, the following transformation is done to (1),(2) along all directions for which a PML is desired:

$$\partial_x \to \left(1 + i\frac{\sigma_x}{\omega}\right)^{-1} \partial_x,$$
$$\partial_y \to \left(1 + i\frac{\sigma_y}{\omega}\right)^{-1} \partial_y,$$
$$\partial_z \to \left(1 + i\frac{\sigma_z}{\omega}\right)^{-1} \partial_z.$$

When the transformations are applied in all directions, we get a system of PDEs that describe the Maxwell equations with PML boundaries at all sides (see appendix A for the derivation). Such a system takes the following form (when we introduce auxiliary differential equations for the integration terms):

$$\begin{aligned}
\partial_t \mathbf{H} &= -\mu^{-1}(\nabla \times (\mathbf{E} + \mathbf{P})) - M_+ \mathbf{H} + \mathbf{R}, \\
\partial_t \mathbf{E} &= \quad \epsilon^{-1} (\nabla \times (\mathbf{H} + \mathbf{Q})) - M_+ \mathbf{E} + \mathbf{S}, \\
\partial_t \mathbf{P} &= \mathrm{diag}(\sigma)\mathbf{E}, \\
\partial_t \mathbf{Q} &= \mathrm{diag}(\sigma)\mathbf{H}, \\
\partial_t \mathbf{R} &= M_* \mathbf{H}, \\
\partial_t \mathbf{S} &= M_* \mathbf{E},
\end{aligned}$$

with $M_+ = \begin{pmatrix} \sigma_y+\sigma_z & 0 & 0 \\ 0 & \sigma_x+\sigma_z & 0 \\ 0 & 0 & \sigma_x+\sigma_y \end{pmatrix}$ and $M_* = \begin{pmatrix} \sigma_y\sigma_z & 0 & 0 \\ 0 & \sigma_x\sigma_z & 0 \\ 0 & 0 & \sigma_x\sigma_y \end{pmatrix}$.

In order to obtain a system with one or more PML boundaries discarded, we can simply set the corresponding σ to zero. An additional constraint is required for consistency, namely: $\mathbf{P}_{t=0} \equiv \mathbf{Q}_{t=0} \equiv \mathbf{R}_{t=0} \equiv \mathbf{S}_{t=0} \equiv 0$. Under this extra condition, if all the transformations are discarded ($\sigma_x \equiv \sigma_y \equiv \sigma_z \equiv 0$), the transformed system reduces to the traditional Maxwell equations.

The given transformations change wave-like solutions to exponentially decaying waves in the regions where $\sigma > 0$. Therefore, we will choose our σ's such that they are greater than zero outside our region of interest. A precise definition will be given in Section 3.3.

Now that we have the final form of our system of PDEs, we can start solving it numerically.

3 Spatial discretization

The first step in solving the system of PDEs is a discretization the curl operators in the Maxwell operator A, effectively turning our system of PDEs into a system of ODEs. For this, we use the well known Yee scheme [7] (a finite-difference discretization). We test the order of convergence of our Maxwell operator A, which according to the Yee scheme should be two. Finally we perform a test to see if our PML regions are working.

3.1 Yee scheme

The main idea of the Yee scheme is that the grids for the **H** and **E** fields are staggered; the grid points where the fields are defined are shifted half a grid step size with respect to each other. The entire numerical domain $\Omega = [a_x, b_x] \times [a_y, b_y] \times [a_z, b_z]$ is divided into respectively n_x, n_y and n_z pieces in every direction, creating a total of $n_x \cdot n_y \cdot n_z$ cubic cells. According to the Yee scheme spatial discretization, the electric field **E** is evaluated on the edges and the magnetic field **H** on the center of the faces of a given cell, as illustrated in Figure 1.

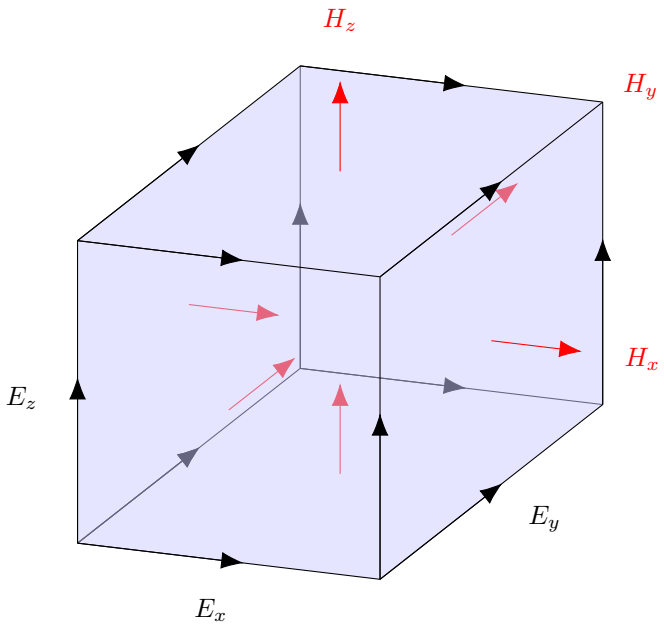

Figure 1: Position of the vector components of the electric and magnetic field with respect to a cubic cell.

With the current setup, there are a total of $(n_x+1)(n_y+1)n_z$ E_z-components, because E_z resides on the edges. For convenience, we add an artificial component in the z-direction. Now there are $N = (n_x + 1)(n_y + 1)(n_z + 1)$ components. This is done analogously for all other directions such that all vector components of **H** and **E** have N components. As the (vector) variables contain information about points on a 3D grid, it is important to define how to enumerate all its components; firstly we iterate over the x-direction, secondly over the y-direction and finally over the z-direction:

$$E_z = (E_z(1,1,1),\ E_z(2,1,1),\ \ldots E_z((nx+1),1,1),\ E_z(1,2,1),\ \ldots)^T .$$

To compute a derivative of one of the variables $E_x, H_x, \ldots$, we apply a differential operator to it. These operators, given the chosen ordering and the spatial discretization, have the following

form:

$$\begin{aligned}
\delta_{x,E} &= I_z \otimes I_y \otimes D_{x,E}, & \delta_{x,H} &= I_z \otimes I_y \otimes D_{x,H},\\
\delta_{y,E} &= I_z \otimes D_{y,E} \otimes I_x, & \delta_{y,H} &= I_z \otimes D_{y,H} \otimes I_x,\\
\delta_{z,E} &= D_{z,E} \otimes I_y \otimes I_x, & \delta_{z,H} &= D_{z,H} \otimes I_y \otimes I_x.
\end{aligned}$$

Here I_x, I_y en I_z are the identity matrices of size $n_x + 1$, $n_y + 1$ and $n_z + 1$ respectively, $\otimes$ denotes the Kronecker product and $D_{i,j}$ are 1D operators defined as follows:

$$D_{x,E} = \frac{1}{h_x} \left.\begin{pmatrix} -1 & 1 & 0 & \cdots \\ 0 & -1 & 1 & \ddots \\ 0 & 0 & -1 & \ddots \\ \vdots & \vdots & \ddots & \ddots \end{pmatrix}\right\} n_x + 1,$$

$$D_{x,H} = \frac{1}{h_x} \left.\begin{pmatrix} 1 & 0 & 0 & \cdots \\ -1 & 1 & 0 & \cdots \\ 0 & -1 & 1 & \ddots \\ \vdots & \vdots & \ddots & \ddots \end{pmatrix}\right\} n_x + 1.$$

Note that these differential operators correspond to a central difference scheme because of the staggering of the grid (and the fact that the spacial derivatives of one field contribute only to the time derivative of the other). Figure 2 gives an explanation of the difference between the given matrices. The remaining differential operators are defined analogously. A simple relation holds: $D_{x,H} = -D_{x,E}^T$. From this it follows directly that: $\delta_{x,H} = -\delta_{x,E}^T$. Thus the discretized curl operators take the form:

$$\nabla_E = \begin{pmatrix} 0 & -\delta_{z,E} & \delta_{y,E} \\ \delta_{z,E} & 0 & -\delta_{x,E} \\ -\delta_{y,E} & \delta_{x,E} & 0 \end{pmatrix}, \qquad \nabla_H = \begin{pmatrix} 0 & \delta_{z,E}^T & -\delta_{y,E}^T \\ -\delta_{z,E}^T & 0 & \delta_{x,E}^T \\ \delta_{y,E}^T & -\delta_{x,E}^T & 0 \end{pmatrix}.$$

and the discretized Maxwell operator with the PML conditions reads:

$$\tilde{A} = \begin{pmatrix} -M_+ & -\mu^{-1}\nabla_E & -\mu^{-1}\nabla_E & 0 & I & 0 \\ \epsilon^{-1}\nabla_H & -M_+ & 0 & \epsilon^{-1}\nabla_H & 0 & I \\ 0 & \mathrm{diag}(\sigma) & 0 & 0 & 0 & 0 \\ \mathrm{diag}(\sigma) & 0 & 0 & 0 & 0 & 0 \\ -M_* & 0 & 0 & 0 & 0 & 0 \\ 0 & -M_* & 0 & 0 & 0 & 0 \end{pmatrix}.$$

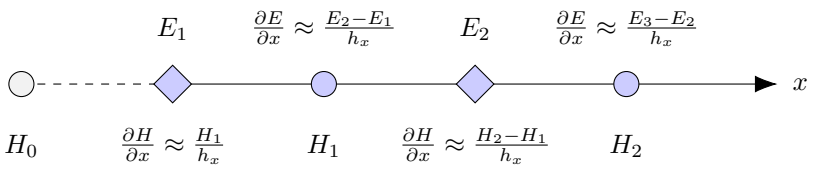

Figure 2: A single line from the grid. H_0 is a ghost cell; the value in this point is set to zero.

3.2 Numerical test of the convergence order

We perform a simple numerical check to confirm that our spatial discretization indeed has convergence order two. We consider a bounded region where $\epsilon \equiv \mu \equiv 1$ (the uniform case)

without the PML absorbing boundary conditions. We choose test functions:

$$
\begin{aligned}
H_x^{\text{in}} &= \sin(c_1 x)\sin(c_2(y+h_y/2))\sin(c_3(z+h_z/2)),\\
H_y^{\text{in}} &= \sin(c_1(x+h_x/2))\sin(c_2 y)\sin(c_3(z+h_z/2)),\\
H_z^{\text{in}} &= \sin(c_1(x+h_x/2))\sin(c_2(y+h_y/2))\sin(c_3 z),\\
E_x^{\text{in}} &= \sin(c_1(x+h_x/2))\sin(c_2 y)\sin(c_3 z),\\
E_y^{\text{in}} &= \sin(c_1 x)\sin(c_2(y+h_y/2))\sin(c_3 z),\\
E_z^{\text{in}} &= \sin(c_1 x)\sin(c_2 y)\sin(c_3(z+h_x/2),
\end{aligned}
$$

and compare their analytical derivative with their numerical approximations on increasingly finer grids. Here $c_1 = 2\pi$ and $c_2 = c_3 = \pi$. The shifts with half the grid size are necessary due to the staggering of the grid. The error is computed with the L_2-norm of all the components. The amount of cells in each direction is varied between 10 and 100. In Figure 3 the results of this test are presented. For comparison, a line with slope 2 is plotted alongside the error plots. We conclude from this figure that the order of convergence of our spatial discretization is indeed 2. Here y represents all the components of the magnetic and electric field, but not the PML

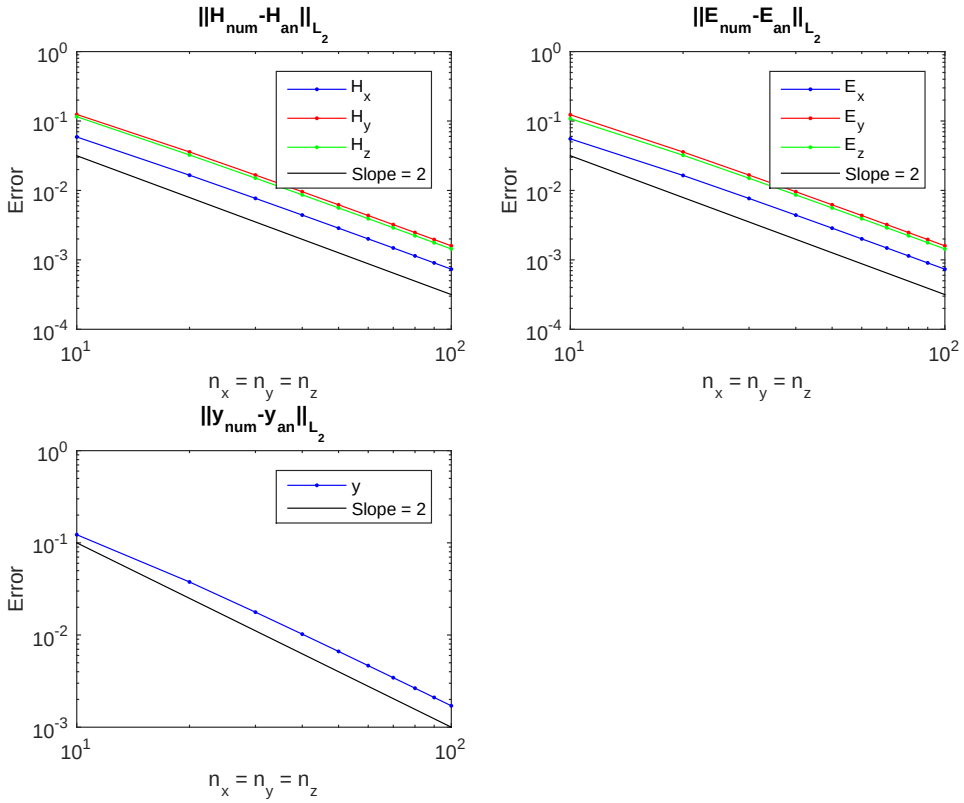

Figure 3: The errors of all the components compared with a line with slope 2.

variables.

3.3 Definition of μ, ϵ and σ

When a PML absorbing boundary is present, the domain Ω, where the Maxwell equations are solved numerically, contains a physical domain (our region of interest) surrounded by a PML region as shown in Figure 4. Everywhere in Ω we take $\mu = 1$ and ϵ varies according to the photonic crystal structure. For the setup shown in Figure 4 we take $\epsilon_1 = 1$ and $\epsilon_2 = 8.9$. Also, we require $\sigma_w \not\equiv 0$ due to the PML conditions for points outside the physical domain in the

w-direction ($w = x, y$). In the analytical case, any choice of $\sigma_w > 0$ would suffice. However, because of numerical considerations we want it to increase gradually, rather than discontinuously. We take our σ_x as follows (and σ_y analogously):

$$\sigma_x(x) = \begin{cases} (x - a_{x_0})^2, & x \in [a_x, a_{x_0}) \\ 0, & x \in [a_{x_0}, b_{x_0}] \\ (x - b_{x_0})^2, & x \in (b_{x_0}, b_x] \end{cases}.$$

In our tests presented in this report, we have a PML only in the x- and y-directions. The z-direction will not require a PML, because in this direction all waves will be reflected back by a photonic crystal. Therefore no significant reflections occur at the boundary.

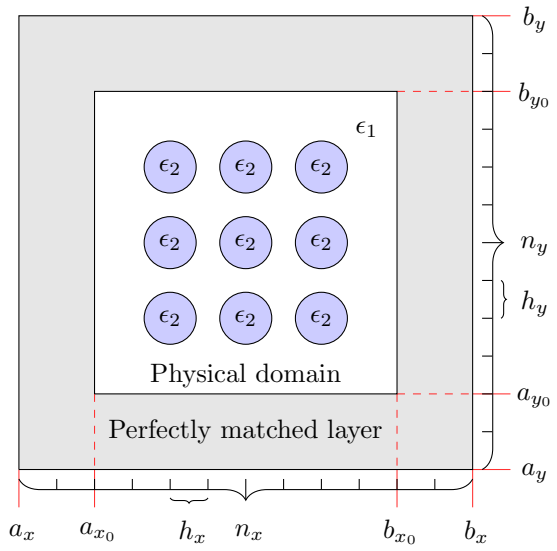

Figure 4: A 2D slice of the numerical domain with a photonic crystal structure.

3.4 Numerical test for numerical reflections of PML

To check whether the PML regions works correctly we perform a simple test. We compare the E_z field for a region with PML and without PML, with $\epsilon \equiv 1$. Our computational domain for the test with PML is $[0, 5] \times [0, 5] \times [0, 3]$ with our region of interest being $[1, 4] \times [1, 4] \times [0, 3]$ (there is no PML region in the z-direction). When we consider the computational domain without PML regions, we should ensure that the waves do not get reflected at the x- and y-boundaries. Therefore, we make our computational domain three times as big in those directions and set it to: $[-2, 7] \times [-2, 7] \times [0, 3]$. We set the initial condition for E_z to be a Gaussian pulse and zero for all the other components of both fields. The E_z field at a time $T = 2$ is shown in Figure 5.

In this figure we see that the pulse decays in the PML region and that the two cases look qualitatively the same int the region of interest.

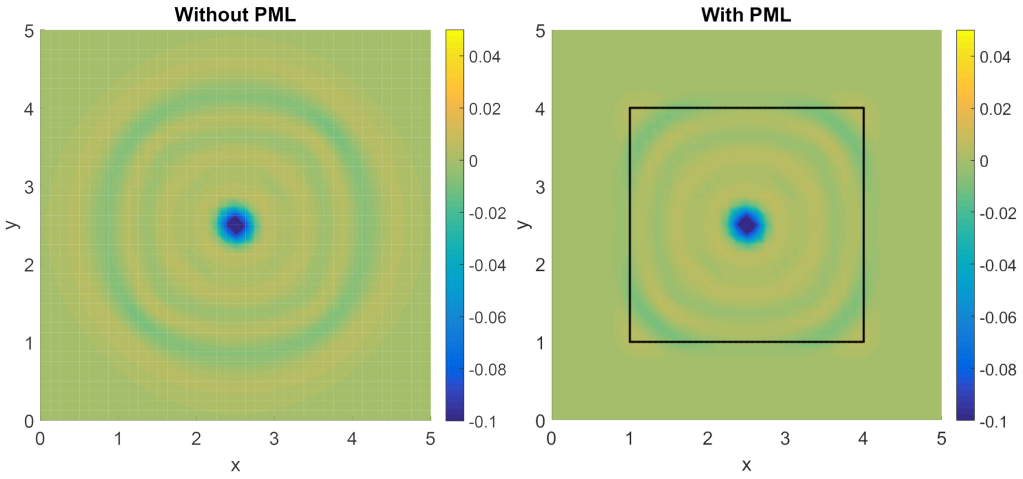

Figure 5: Numerical check for a photonic crystal with and without the PML region.

4 Time integrators

To solve the linear system of (3), a variety of different methods can be used. We will focus on using exponential time integration, which relies on computing approximations to the exponent matrix to get numerical results. We will employ the `expokit` package to calculate a reference solution. We will focus on using Krylov shift-and-inverse (Krylov SAI) exponential time integration, which allows for different choices of solvers for linear systems. For this we will use a direct solver and two iterative solvers: the generalized minimal residual method (GMRES) and the biconjugate gradient stabilized method (BiCGSTAB). Finally we will also use a time-stepping method, namely the implicit trapezoidal rule method (ITR).

4.1 `expokit` package

One of the exponential time integration methods we have used is the `expokit` Matlab package [9]. This package can be employed to compute actions of the matrix exponential. We will compare the performance of Krylov SAI with this method.

4.2 Krylov SAI

To compute the exponential of a matrix the Krylov SAI method [10] can also be used. First we will explain the Krylov subspace method (without SAI) and argue why this method is not always suitable. Then we will describe the incorporation of SAI.

We consider the solution of 3 given by the matrix exponential:

$$\mathbf{y}(t) = \mathrm{e}^{tA}\mathbf{v}. \tag{4}$$

Here A is the result of an FDTD discretization, this is the same matrix as we used before for the Maxwell equations. The Krylov subspace of A and $\mathbf{v}$ is defined as:

$$\mathcal{K}_m(A, \mathbf{v}) = \mathrm{span}\{\mathbf{v}, A\mathbf{v}, A^2\mathbf{v}, \ldots, A^{m-1}\mathbf{v}\}.$$

In the Krylov subspace method the orthogonal basis $\{\mathbf{v}_1, \mathbf{v_2}, \ldots, \mathbf{v}_m\}$ of $\mathcal{K}_m(A, \mathbf{v})$ is computed and stored column wise in the matrix $V_m = [\mathbf{v}_1\mathbf{v_2}\ldots\mathbf{v}_m]$. We compute this matrix with the Arnoldi (or Arnoldi SAI) process. The matrix V_m satisfies the Arnoldi relation:

$$AV_m = V_{m+1}H_{m+1,m}. \tag{5}$$

Here $H_{m+1,m} \in \mathbb{R}^{m+1,m}$ is an upper Hessenberg matrix. This is a matrix with all zero elements below the subdiagonal, so $h_{i,j} = 0$ for $i > j + 1$. Define $H_{m,m}$ as the first m rows of $H_{m+1,m}$, then we can rewrite (5) as:

$$AV_m = V_m H_{m,m} + \mathbf{v}_{m+1}h_{m+1,m}\mathbf{e}_m^T, \tag{6}$$

where $\mathbf{e}_m^T$ is the canonical vector $[0, \ldots, 0, 1]$ of length m. This relation tells us that the matrix A times any vector of the basis of the Krylov subspace is another vector in the Krylov subspace plus a multiple of the next basis vector $\mathbf{v}_{m+1}$. We can use this to approximate $\mathbf{y}(t)$ of (4). We set $\beta = \|\mathbf{v}\|$ and choose $\mathbf{v}_1 = \mathbf{v}/\beta$ as the first basis vector and V_m and $H_{m,m}$ can be computed using the Arnoldi process. Hence, the approximation $\mathbf{y}_m(t)$ of $\mathbf{y}(t)$ can be used [11,12,13]:

$$\begin{aligned}\mathbf{y}(t) &= \mathrm{e}^{tA}\mathbf{v} = \mathrm{e}^{tA}V_m\beta\mathbf{e}_1,\\ \mathbf{y}_m(t) &= V_m\mathrm{e}^{tH_{m,m}}\beta\mathbf{e}_1.\end{aligned}$$

Here we apply the Arnoldi relation to e^{tA} instead of A. This is possible because the matrix exponential is just a polynomial function in A. This is a good approximation as long as the term $\mathbf{v}_{m+1}h_{m+1,m}\mathbf{e}_m^T$ is small. Computing $\mathbf{y}_m(t)$ is much faster than computing $\mathbf{y}(t)$ because $m \ll n$ (n being the size of A) and m is usually around one hundred. The matrix $\mathrm{e}^{tH_{m,m}}$ can therefore

be calculated using standard solvers for the matrix exponential.
The stopping criterion is based on the residual of the approximation with respect to the ODE:

$$\mathbf{r}_m(t) = \mathbf{y}'_m(t) - A\mathbf{y}_m(t).$$

By taking the derivative of $\mathbf{y}_m(t)$ and using equation (6) the residual is

$$\mathbf{r}_m(t) = -\mathbf{v}_{m+1} h_{m+1,m} \mathbf{e}_m^T \mathrm{e}^{-tH_{m,m}} \beta \mathbf{e}_1.$$

When the norm of the residual gets smaller than some given tolerance the Arnoldi process is stopped.
One problem with the Krylov subspace method can exhibit a slow convergence to $\mathbf{y}t$. This is because the matrix can have a wide spectrum with both small and large eigenvalues. The eigenvalues of $H_{m,m}$ give a good approximation for the larger eigenvalues of A, but these are not important for the computation of the matrix exponential. Therefore we use the shifted and inverse matrix of A: $(I-\gamma A)^{-1}$, to build the Krylov subspace, with $\gamma > 0$ a parameter that can be chosen. The Arnoldi SAI relation holds:

$$\begin{aligned}(I-\gamma A)^{-1} V_m &= V_{m+1} \widetilde{H}_{m+1,m}, \\ &= V_m \widetilde{H}_{m,m} + \mathbf{v}_{m+1} \tilde{h}_{m+1,m} \mathbf{e}_m^T. \end{aligned} \tag{7}$$

We can now use the same approximation $\mathbf{y}_m(t) = V_m \mathrm{e}^{tH_{m,m}} \beta \mathbf{e}_1$ with [14,15]

$$H_{m,m} = \frac{1}{\gamma}\left(I - \widetilde{H}_{m,m}^{-1}\right).$$

The algorithm for the Krylov SAI method is given in algorithm 1 (see [16]). Computing $(I-\gamma A)^{-1}$ directly is not possible for a large A such as our Maxwell operator. Therefore we solve the system $(I-\gamma A)\,\mathbf{w} = \mathbf{v}_j$. This can be done by either a direct or iterative solver (where the latter allows for preconditioning). Later on we will look at some preconditioning methods and compare them.
The stopping criterion is now more important because it also affects the amount of times we have to solve the system $(I-\gamma A)\,\mathbf{w} = \mathbf{v}_j$. Computing the residual of the Krylov SAI method is a bit more complicated than in the conventional Krylov subspace method. First we rewrite the Arnoldi SAI relation of (7) as:

$$AV_m = V_m H_{m,m} + \frac{1}{\gamma}(I-\gamma A)\,\mathbf{v}_{m+1} \tilde{h}_{m+1,m} e_m^T \widetilde{H}_{m,m}^{-1}.$$

Then we can use the same definition for the residual and it follows that [17]:

$$\mathbf{r}_m(t) = -\frac{1}{\gamma}(I-\gamma A)\,\mathbf{v}_{m+1} \tilde{h}_{m+1,m} \mathbf{e}_m^T \widetilde{H}_{m,m}^{-1} \mathrm{e}^{tH_{m,m}} \beta \mathbf{e}_1.$$

4.2.1 Direct solver

The easiest method of solving linear systems is by using a direct solver. In our case, we will use a sparse LU factorization using the UMFPACK in Matlab. This method factorizes the a matrix A into matrices L, U, P, Q and R such that:

$$\begin{aligned} PR^{-1}AQ &= LU, \\ \Rightarrow A &= RP^{-1}LUQ^{-1}. \end{aligned}$$

Here L and U are respectively lower and upper triangular matrices, P and Q are permutation matrices, which are used to minimize the fill-in (the number of non-zero elements in L and U relative to the number of non-zero elements in A) and R is a diagonal matrix. To solve the linear system $A\mathbf{x} = \mathbf{b}$, we can rewrite this as:

$$LU\mathbf{x} = PR^{-1}\mathbf{b}Q = \tilde{\mathbf{b}}.$$

Now we can solve $LU\mathbf{x} = \tilde{\mathbf{b}}$ by using forward and backward substitution. For large sparse matrices, coming from discretization of 3D PDEs, LU factorization is an expensive process, in terms of computation time and in memory.

Algorithm 1 Arnoldi SAI process for $y = \mathrm{e}^{-tA}v$

$\beta = \|\mathbf{v}\|, \mathbf{v}_1 = \mathbf{v}/\beta$
for $j = 1, 2, \ldots, m$ **do**
  $\mathbf{w} = (I - \gamma A)^{-1} \mathbf{v}_j$
  for $i = 1, 2, \ldots, j$ **do**
    $H_{i,j} = \mathbf{w}^T \cdot \mathbf{v}_i$
    $\mathbf{w} = \mathbf{w} - H_{i,j}\mathbf{v}_i$
  end for
  $H_{j+1,j} = \|\mathbf{w}\|$
  $\underline{H} = \left(I - H^{-1}_{1:j,1:j}\right)/\gamma$
  $C = \|(I - \gamma A)\mathbf{w}\|$
  $u = \mathrm{expm}(-t\underline{H})\mathbf{e}_1$
  $\mathbf{r}_m(t) = C\mathbf{e}_j^T H^{-1}_{1:j,1:j} u/\gamma$
  resnorm $= \|\mathbf{r}_m(t)\|$
  if resnorm $<$ toler **then** stop
  end if
  $\mathbf{v}_{j+1} = \mathbf{w}/H_{j+1,j}$
end for
$\mathbf{y} = V_{1:j}\beta u$

4.2.2 GMRES and BiCGSTAB iterative solvers

For large linear systems direct methods are too expensive. Therefore we will use an iterative process to solve the system $(I - \gamma A)\,\mathbf{w} = \mathbf{v}_j$ in the Arnoldi SAI process. We start with an initial guess $\mathbf{w}_0$ and for each iteration m we take a search direction $\mathbf{z}_m$, such that:

$$\mathbf{w}_m = \mathbf{w}_0 + \mathbf{z}_m.$$

The question now is: how can we find a good search direction? One option is to choose $\mathbf{z}_m$ from the Krylov subspace of $\widetilde{A} = I - \gamma A$ and $\mathbf{r}_0 = \mathbf{v}_j - \widetilde{A}\mathbf{w}_0$. Note that the Krylov subspace for A and $\widetilde{A}$ are identical.
Now suppose that $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m\}$ is a basis of the Krylov subspace and let $V_m = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_m]$. Then for every element $\mathbf{z}_m$ from $\mathcal{K}_m(A, \mathbf{r}_0)$ there is a vector $\mathbf{x}$ such that $\mathbf{z}_m = V_m\mathbf{x}$. So the problem of finding a search direction in the Krylov subspace can be reduced to finding a vector $\mathbf{x}$. To find the best search direction, the solver GMRES finds $\mathbf{x}$ such that the residual $\mathbf{r}_m$ is minimal. GMRES uses the Arnoldi process.

Another Krylov subspace method which we will use is BiCGSTAB [18]. In this method we change the Arnoldi relation (5) in such a way that $H_{m,m}$ is tridiagonal. The new relation is:

$$AV_m = W_m H_{m,m}.$$

The orthogonal columns of W_m spans a second Krylov subspace, namely $\mathcal{K}_m(A^T, \tilde{\mathbf{r}})$, where $\tilde{\mathbf{r}}$ is an arbitrary vector. In BiCGSTAB, the search direction is chosen such that the residual vector $\mathbf{r}_m$ is orthogonal to the second Krylov subspace: $W_m^T\mathbf{r}_m = 0$.

4.3 ITR

To compare the performance of the Krylov subspace methods described above with the conventional time stepping integration methods, we include the ITR method in our tests. At every time step k, ITR calculates the vector $\mathbf{y}_k \approx \mathbf{y}(k\tau)$ using the following formula:

$$\mathbf{y}_k = \mathbf{y}_{k-1} + \frac{\tau}{2}\left(A\mathbf{y}_{k-1} + A\mathbf{y}_k\right).$$

Here τ is the time step. To obtain $\mathbf{y}_k$, we need to solve the system:

$$\left(I - \frac{\tau}{2}A\right)\mathbf{y}_k = \left(I + \frac{\tau}{2}A\right)\mathbf{y}_{k-1}.$$

On coarse grids we can use LU factorization to solve this system. However, for larger grids, the same problems arise as in solving the original system (3). To do this, we can again use iterative methods.

5 Preconditioning

To obtain accurate results, we need to use relatively fine grids. However, this leads to a large size of the matrix and thus to high computation time and memory costs. We can combat this by using two-sided preconditioning. This means that, instead of the system $Ax = b$, we solve an equivalent system $\tilde{A}\tilde{x} = \tilde{b}$, on which iterative solvers converge faster. For this, we use a preconditioner matrix $M = M_1 M_2$ such that M resembles A, but linear systems with M_1 and M_2 are easy to solve. In the case of two-sided preconditioning, we get the system:

$$\begin{aligned} M_1^{-1} b &= M_1^{-1} A M_2^{-1} M_2 x, \\ \tilde{A} &= M_1^{-1} A M_2^{-1}, \\ \tilde{x} &= M_2 x, \\ \tilde{b} &= M_1^{-1} b. \end{aligned} \tag{8}$$

Since our solver should work for matrices of the form $I - \gamma A$ for some positive γ, we need to find matrices M_1 and M_2 such that the product approximates $I - \gamma A$. We use preconditioner matrices based on splitting, i.e., we define A_1 and A_2 such that $A = A_1 + A_2$ and $M_i = I - \gamma A_i$, so that

$$I - \gamma A \approx (I - \gamma A_1)(I - \gamma A_2). \tag{9}$$

5.1 Alternating Direction Implicit splitting

The first splitting we will look at is based on the Alternating Direction Implicit (ADI) method (see e.g. [8]). The idea is to split the curl operator in the original Maxwell equations into two components:

$$B_1 = \begin{pmatrix} 0 & 0 & \delta_y \\ \delta_z & 0 & 0 \\ 0 & \delta_x & 0 \end{pmatrix}, B_2 = \begin{pmatrix} 0 & \delta_z & 0 \\ 0 & 0 & \delta_x \\ \delta_y & 0 & 0 \end{pmatrix}, \nabla = B_1 - B_2.$$

This means that, if we ignore the σ-terms in our original matrix, we have

$$A = \begin{pmatrix} 0 & \frac{1}{\mu}(B_2 - B_1) \\ \frac{1}{\epsilon}(B_1 - B_2) & 0 \end{pmatrix}.$$

Now we can choose A_1 and A_2 as follows:

$$A_1 = \begin{pmatrix} 0 & \frac{1}{\mu} B_2 \\ \frac{1}{\epsilon} B_1 & 0 \end{pmatrix}, A_2 = \begin{pmatrix} 0 & -\frac{1}{\mu} B_1 \\ -\frac{1}{\epsilon} B_2 & 0 \end{pmatrix}.$$

The diagonal terms of A have been evenly divided amongst A_1 and A_2. Lastly, the curl-PML terms are split according to the description given above and all other terms are evenly divided as well:

$$A_1 = \begin{pmatrix} -M_+/2 & \mu^{-1} B_2 & \mu^{-1} B_2 & 0 & I/2 & 0 \\ \epsilon^{-1} B_1 & -M_+/2 & 0 & \epsilon^{-1} B_1 & 0 & I/2 \\ 0 & \mathrm{diag}(\sigma)/2 & 0 & 0 & 0 & 0 \\ \mathrm{diag}(\sigma)/2 & 0 & 0 & 0 & 0 & 0 \\ -M_*/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -M_*/2 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$A_2 = \begin{pmatrix} -M_+/2 & -\mu^{-1} B_1 & -\mu^{-1} B_1 & 0 & I/2 & 0 \\ -\epsilon^{-1} B_2 & -M_+/2 & 0 & -\epsilon^{-1} B_2 & 0 & I/2 \\ 0 & \mathrm{diag}(\sigma)/2 & 0 & 0 & 0 & 0 \\ \mathrm{diag}(\sigma)/2 & 0 & 0 & 0 & 0 & 0 \\ -M_*/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & -M_*/2 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

5.2 Field splitting

The second splitting is based on separation of the contributions of the **H** and **E** fields into the two matrices A_1 and A_2. In other words, all components involved in the time integration of **H** are in one matrix and all other components in the other:

$$A_1 = \begin{pmatrix} -M_+ & -\mu^{-1}\nabla_E & -\mu^{-1}\nabla_E & 0 & I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \mathrm{diag}(\sigma) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -M_* & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix},$$

$$A_2 = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \epsilon^{-1}\nabla_H & -M_+ & 0 & \epsilon^{-1}\nabla_H & 0 & I \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \mathrm{diag}(\sigma) & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -M_* & 0 & 0 & 0 & 0 \end{pmatrix}.$$

6 Numerical tests

To compare the different solvers and splitting methods we create a test case. The total domain is $[0,5] \times [0,5] \times [0,3]$ and the region of the photonic crystal is $[1,4] \times [1,4] \times [0,3]$. So the PML region is $[0,1]$ and $[4,5]$ for both the x- and y-direction. Inside the photonic crystal we place 27 spheres ($3 \times 3 \times 3$) with radius 0.4. Inside these spheres the electrical permittivity ϵ is set to 8.9 and ϵ is1 outside the spheres. The permeability μ is taken as 1 everywhere in the domain. The initial values for the magnetic and electric field are all zero except of E_z, which reads

$$E_z = \mathrm{e}^{-30((x-x_0)^2+(y-y_0)^2+10(z-z_0)^2)}. \tag{10}$$

Here x_0, y_0 and z_0 are at the center of the domain, so $x_0 = y_0 = 2.5$ and $z_0 = 1.5$.

6.1 Computation time

We compare `expokit`, ITR and Krylov SAI methods. For the linear system in the Krylov SAI method we use different solvers: the direct sparse solver, GMRES and BiCGSTAB. In GMRES and BiCGSTAB we use none, ADI- or field-splitting preconditioners. When using a preconditioner we also look at the influence of γ on the computation time by taking two different values: $\gamma = 0.012$ and $\gamma = 0.035$. The results are presented in Tables 1–6. When there is an '–' in the table, it means that we were not able to compute that value due to hardware limitations.

We compare the CPU times for all of the solvers. This is done for the final physical time $T = 1$. Then for larger values of T we only use the solvers that are faster or equally fast than `expokit` to check if they also perform well for higher final times T. The following conclusions can be drawn from the tests

- ITR is not faster than `expokit` or Krylov SAI in any case.
- A direct solver in Krylov SAI is both slower and less efficient on finer grids than GMRES and BiCGSTAB.
- Using a preconditioner is effective and Field splitting is faster than ADI-splitting.
- $\gamma = 0.012$ is more efficient than $\gamma = 0.035$ (also for higher T values). See also Section 6.3.
- BiCGSTAB is faster than GMRES.
- BiCGSTAB and GMRES, both with Field splitting, are faster than `expokit` for the final time $T = 1$. However, for larger final times T `expokit` is faster than either one of them.
- These conclusions hold for both choices of ϵ. This means that we can use the best solver (in computation time) for different cases of ϵ, like the line defect in chapter 7.1.

6.1.1 Results with uniform epsilon

	Solver	Mesh $[n_x, n_y, n_z]$			
		$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.25	1.15	16.6	229
	ITR	0.39	3.03	73.7	831
$\gamma = 0.012$	Direct	0.13	1.41	68.9	–
	GMRES without preconditioning	0.44	4.11	60.0	611
	GMRES with ADI-splitting	0.22	1.72	25.4	454
	GMRES with field-splitting	0.14	0.85	11.9	152
	BiCGSTAB without preconditioning	0.25	1.74	30.3	332
	BiCGSTAB with ADI-splitting	0.16	0.75	9.98	131
	BiCGSTAB with field-splitting	0.12	0.57	8.07	105
$\gamma = 0.035$	Direct	0.10	1.19	67.9	–
	GMRES without preconditioning	0.91	5.69	136	2003
	GMRES with ADI-splitting	0.16	1.29	28.9	622
	GMRES with field-splitting	0.11	0.64	16.4	318
	BiCGSTAB without preconditioning	0.27	2.63	56.0	809
	BiCGSTAB with ADI-splitting	0.12	0.65	12.8	209
	BiCGSTAB with field-splitting	0.08	0.46	11.0	203

Table 1: The CPU times in seconds for different methods for $\epsilon = 1$ everywhere. Final time $T = 1$.

6.1.2 Results with 3D epsilon (spheres)

	Solver	Mesh $[n_x, n_y, n_z]$			
		$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.26	1.16	17.3	234
	ITR	0.46	2.85	69.1	812
$\gamma = 0.012$	Direct	0.13	1.30	67.9	–
	GMRES without preconditioning	0.40	3.49	57.8	796
	GMRES with ADI-splitting	0.22	1.42	25.6	479
	GMRES with field-splitting	0.13	0.66	11.7	168
	BiCGSTAB without preconditioning	0.22	1.60	29.3	388
	BiCGSTAB with ADI-splitting	0.16	0.64	9.82	147
	BiCGSTAB with field-splitting	0.11	0.47	8.11	113
$\gamma = 0.035$	Direct	0.10	1.22	66.6	–
	GMRES without preconditioning	0.81	6.51	113	2617
	GMRES with ADI-splitting	0.18	1.29	28.9	907
	GMRES with field-splitting	0.11	0.65	15.2	393
	BiCGSTAB without preconditioning	0.24	2.19	51.7	930
	BiCGSTAB with ADI-splitting	0.11	0.61	11.9	276
	BiCGSTAB with field-splitting	0.09	0.47	10.5	233

Table 2: The CPU times in seconds for different methods for $\epsilon = 8.9$ inside the spheres and 1 elsewhere. Final time $T = 1$.

	Mesh $[n_x, n_y, n_z]$				
	Solver	$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.25	1.73	33.5	389
$\gamma = 0.012$	GMRES with field-splitting	0.18	1.15	24.0	458
	BiCGSTAB with field-splitting	0.15	0.88	16.6	299
$\gamma = 0.035$	GMRES with field-splitting	0.17	1.52	55.6	2040
	BiCGSTAB with field-splitting	0.18	1.05	36.4	1264

Table 3: The CPU times in seconds for different solving methods for $\epsilon = 8.9$ inside the spheres and 1 elsewhere. Final time $T = 2$.

	Mesh $[n_x, n_y, n_z]$				
	Solver	$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.25	1.82	42.2	625
$\gamma = 0.012$	GMRES with field-splitting	0.25	1.66	40.3	1052
	BiCGSTAB with field-splitting	0.19	1.19	27.8	650

Table 4: The CPU times in seconds for different solving methods for $\epsilon = 8.9$ inside the spheres and 1 elsewhere. Final time $T = 3$.

	Mesh $[n_x, n_y, n_z]$				
	Solver	$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.38	2.29	60.6	845
$\gamma = 0.012$	GMRES with field-splitting	0.29	2.54	67.8	2208
	BiCGSTAB with field-splitting	0.22	1.75	46.4	1284

Table 5: The computation times in seconds for different solving methods for $\epsilon = 8.9$ inside the spheres and 1 elsewhere. Final time $T = 4$.

	Mesh $[n_x, n_y, n_z]$				
	Solver	$[10, 10, 6]$	$[20, 20, 12]$	$[40, 40, 24]$	$[80, 80, 48]$
	`expokit`	0.37	2.82	77.5	1010
$\gamma = 0.012$	GMRES with field-splitting	0.31	3.34	107	4001
	BiCGSTAB with field-splitting	0.46	2.17	71.1	2629

Table 6: The CPU times in seconds for different solving methods for $\epsilon = 8.9$ inside the spheres and 1 elsewhere. Final time $T = 5$.

6.2 Fill-in factors

An important factor which determines the computation time for preconditioned iterative solvers, is the fill-in factor. Let $[L, U]$ be the LU factorization of the preconditioner matrix M. The fill-in factor is defined as

$$\frac{\text{nnz}(L+U)}{\text{nnz}(M)},$$

where nnz() equals the number of non-zero elements in a matrix. Since M is a sparse matrix, we would like to keep the sparsity intact after the factorization (the more non-zero elements in a matrix, the more flops are needed to solve the system). A fill-in factor of 1 is ideal, since this means that there is no fill-in and the sparsity of the matrix remains completely intact. The bigger the fill-in factor, the more time is needed to solve a linear system with M.

In Table 7, the fill-in factors for the ADI preconditioners are given, for each of the matrices M_1 and M_2. We see that the larger the grid, the more fill-in is needed. For field-splitting however, all of the fill-in factors are 1. This gives a considerate advantage to field splitting, especially for larger systems. We can also note that for both preconditioners there is no time dependency for the fill-in factors. This means that for larger time intervals the fill-in is the same.

$[n_x, n_y, n_z]$	$T=1$	$T=2$	$T=3$	$T=4$
$[10, 10, 6]$	1.2582	1.2582	1.2582	1.2582
	1.3004	1.3004	1.3004	1.3004
$[20, 20, 12]$	1.3736	1.3736	1.3736	1.3736
	1.3947	1.3947	1.3947	1.3947
$[40, 40, 24]$	1.4881	1.4881	1.4881	1.4881
	1.4995	1.4995	1.4995	1.4995
$[80, 80, 48]$	1.6006	1.6006	1.6006	1.6006
	1.6244	1.6244	1.6244	1.6244

Table 7: Fill-in factors for ADI-splitting

6.3 Optimal value for γ

In the previous cases we have set $\gamma = 0.012$ or $\gamma = 0.035$ and compared the results of these two values. We check if there are better values for γ than 0.012. Therefore we look for some other values of γ and check the CPU time with BiCGSTAB (because it is faster than GMRES) and field-preconditioning on a $[40, 40, 24]$ grid. If the computation time is close to each other then we look at the $[80, 80, 48]$ grid. The results of this test are presented in Table 8.

γ	$[40, 40, 24]$	$[80, 80, 48]$
0.008	14.03	
0.010	12.82	183.1
0.012	12.38	184.6
0.014	12.54	182.9
0.016	12.50	215.2
0.018	10.60	213.2
0.020	12.82	221.5
0.024	12.85	
0.028	16.00	
0.032	15.44	
0.036	15.10	
0.040	15.07	

Table 8: The CPU times in seconds of the Krylov SAI method for different values of γ with BiCGSTAB.

From Table 8 we conclude that $\gamma = 0.012$ is indeed one of the optimal values for γ. $\gamma = 0.010$ or $\gamma = 0.014$ give the same results for the $[80, 80, 48]$ grid and are therefore equally good. Although $\gamma = 0.018$ has a lower computation time for the $[40, 40, 24]$ grid, it is not faster for the $[80, 80, 48]$ grid. We assume that this is a special case and so is not optimal.
We can do the same test for GMRES. For GMRES the results are presented in Table 9. Here we see that $\gamma = 0.012$ is indeed optimal for GMRES. Also for $\gamma = 0.018$ the same lower computation time occurs for the $[40, 40, 24]$ grid.

γ	$[40, 40, 24]$	$[80, 80, 48]$
0.008	22.30	
0.010	21.37	
0.012	18.72	267.1
0.014	18.20	295.0
0.016	18.74	323.6
0.018	15.64	339.2
0.020	18.86	335.5
0.024	21.28	
0.028	21.95	
0.032	22.80	
0.036	22.67	
0.040	25.96	

Table 9: The CPU times in seconds of the Krylov SAI method for different values of γ with GMRES.

7 Applications

The numerical solution of the Maxwell equations is useful in, for example, the simulation of photonic crystal structures. An important real world application in this field is in a design of electromagnetic waveguides such as optical fibers. Optical fibers are fibers that are used for communication that relies on the transmission of light to send data. In this section we show a few structures that have been successfully simulated with our solver. The initial value is a Gaussian pulse given by (10) in the center of the region of interest. In all cases we consider a variation of a 3D structure of spheres. The shown results are the E_z field plotted in the xy-plane for $z = 1.5$ (the middle of the photonic crystal).

7.1 Line defect

A line defect (see Figure 6) is a structure in which a row of spheres has been removed from the regular lattice of spheres (in our case $3 \times 3 \times 3$ array of spheres). This structure acts as a waveguide; waves tend to travel along the line defect and not in the other directions. As seen in Figure 7, the expected behavior is found by the simulation.

7.2 A closed line defect

We now consider the line defect with spheres added at the ends of the line defect (see Figure 8). As before, the waves are guided, but now they get reflected by the spheres at the end of the line defect. The difference with the previous case is hard to discern, but is there nevertheless; the waves get reflected at the added spheres at the edges (see Figure 9).

7.3 Around the corner

Based on the observations of the first case, one might expect similar behavior for any given path through a photonic crystal. Sadly, this is not the case according to our simulation (see Figures 10, 11); the waves that are guided through the bend are damped a lot.

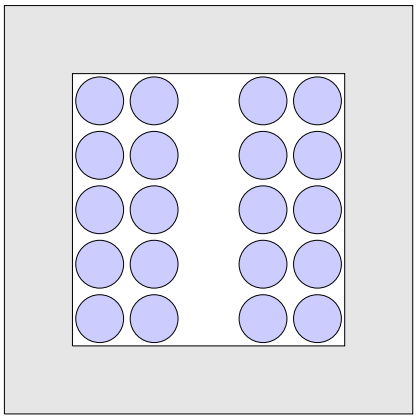

Figure 6: A line defect in the photonic crystal

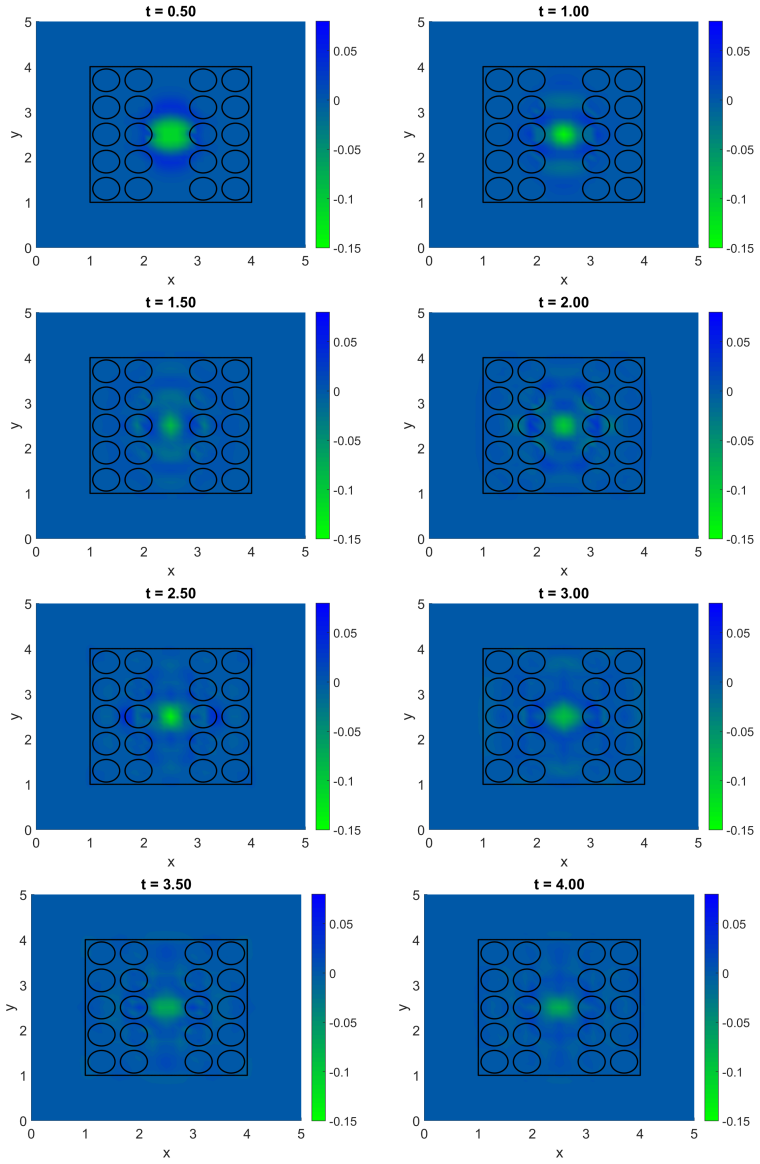

Figure 7: Gaussian pulse initial condition in line defect photonic crystal structure

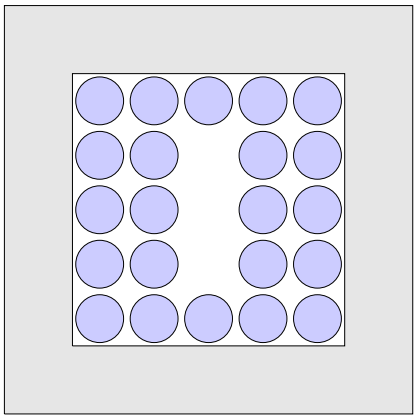

Figure 8: A closed line defect in the photonic crystal

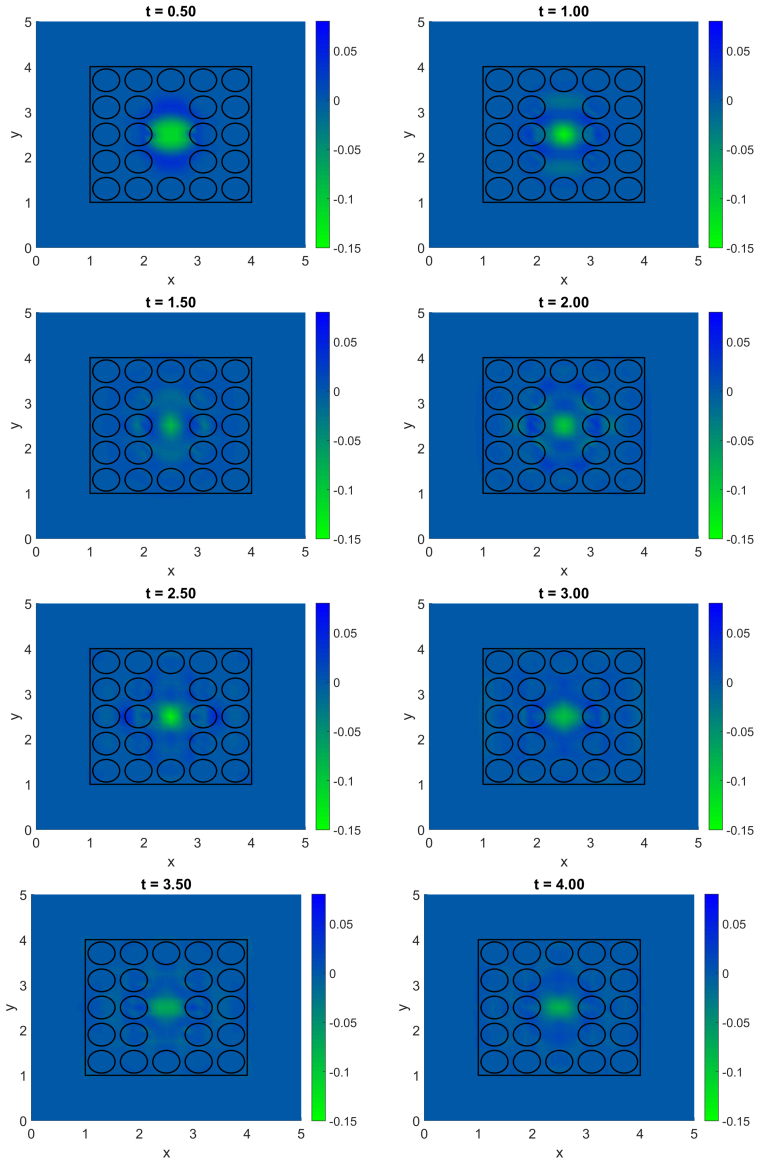

Figure 9: Gaussian pulse initial condition in bounded line defect photonic crystal structure

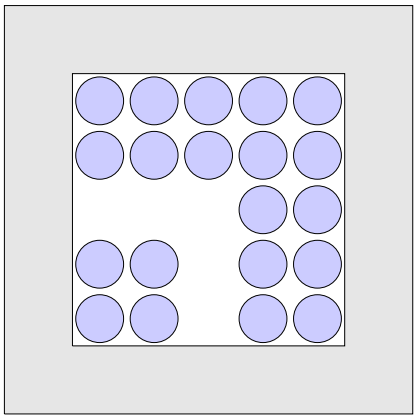

Figure 10: A line defect in two dimensions in the photonic crystal

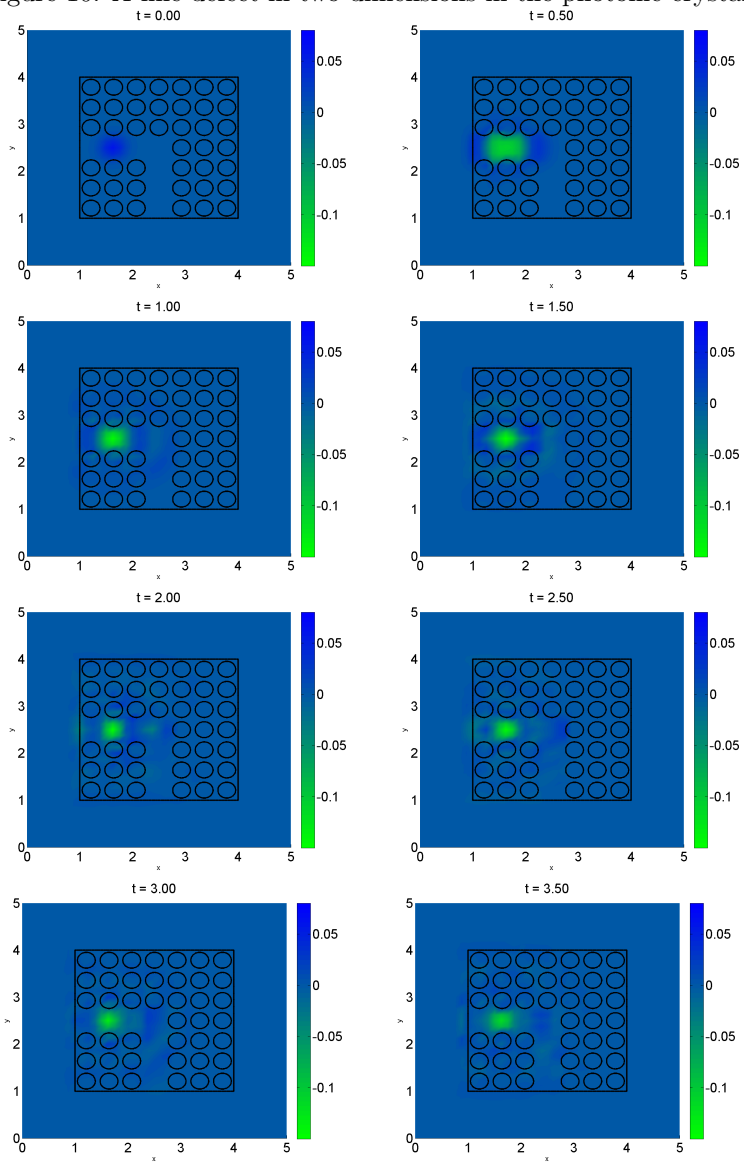

Figure 11: Gaussian pulse initial condition in "around-the-corner" photonic crystal structure

8 Conclusion

We considered time integration methods for solving the time dependent Maxwell equations as arise in modeling of photonic crystal structures.

For the space discretized equations we compared the CPU time for a number of different solvers with the existing `expokit` package. One solver we used is ITR (Implicit Trapezoidal Rule method), whereas all of the other solvers, including `expokit`, are based on approximating matrix exponential actions. The Krylov SAI method was employed with different linear solvers. We compared the performance of a sparse direct solver and iterative solvers GMRES and BiCGSTAB. For GMRES and BiCGSTAB we analysed the influence of preconditioning, namely preconditioners based on the ADI- and field-splitting.

The results show that for low values of the final time T Krylov SAI with BiCGSTAB and field-splitting preconditioner is the fastest method. However for larger final times T, `expokit` is faster than Krylov SAI. ITR was in all cases worse than `expokit` or Krylov SAI. Field-splitting has no fill in whereas the ADI-splitting has a fill in between 1.3 and 1.6, depending on the grid.

References

[1] D.J. Griffiths, *Introduction to Electrodynamics*, Pearson, 4th edition, p. 398–399, 2013.

[2] A. Taflove, S. Hagness, *Computational Electrodynamics: the finite-difference time-domain method*, Artech House, 2nd edition, p. 74–75 & 295–297, 2000.

[3] B. Fornberg, *Some Numerical Techniques for Maxwell's Equations in Different Types of Geometries*, Topics in Computational Wave Propagation, Lecture notes in Computational Science and Engineering 31, Springer Verlag, p. 265–299, 2003. (p. 3)

[4] J. Bérenger, *A Perfectly Matched Layer for the Absorption of Electromagnetic Waves*, J. Comput. Phys., vol. 114, no. 1, p. 185–200, 1994.

[5] W.C. Chew, W. Weedon, *A 3-D Perfectly Matched Medium from Modified Maxwell's Equations with Stretched Corodinates*, IEEE Microwave and Guided Wave Letters, Vol. 7, p. 599–604, 1994.

[6] S.G. Johnson, *Notes on Perfectly Matched Layers (PMLs)*, MIT, 2010.

[7] K.S. Yee, *Numerical Solution of Initial Boundary Value Problems Involving Maxwell's Equations in Isotropic Media*, IEEE Trans. Antennas Propagat. vol. AP-14, p. 302–307, 1966.

[8] M. Hochbruck, T. Jahnke, R. Schnaubelt, *Convergence of an ADI Splitting for Maxwell's Equations*, Numerische Mathematik, vol. 129, issue 3, p. 535–561, 2015.

[9] R.B. Sidje, *`Expokit`: A Software Package for Computing Matrix Exponentials*, ACM Trans. Math. Softw. 24, p. 130–156, 1998.

[10] M.A. Botchev, *Krylov subspace exponential time domain solution of Maxwell's equations in photonic crystal modeling*, Journal of Computational and Applied Mathematics (2015), http://dx.doi.org/10.1016/j.cam.2015.04.022

[11] G. Rodrigue, D. White, *A vector finite element time-domain method for solving Maxwell's equations on unstructured hexahedral grids*, SIAM J. Sci. Comput. 23 (3) (2001) 683706.

[12] M. Hochbruck, A. Ostermann, *Exponential integrators*, Acta Numer. 19 (2010) 209286

[13] N.J. Higham, *The scaling and squaring method for the matrix exponential revisited*, SIAM J. Matrix Anal. Appl. 26 (4) (2005) 11791193

[14] E. Gallopoulos, Y. Saad, Efficient solution of parabolic equations by Krylov approximation methods, SIAM J. Sci. Stat. Comput. 13 (5) (1992) 12361264

[15] J. Niehoff, *Projektionsverfahren zur Approximation von Matrixfunktionen mit Anwendungen auf die Implementierung exponentieller Integratoren* , Mathematisch-Naturwissenschaftlichen Fakultt der Heinrich-Heine-Universitt Dsseldorf, 2006

[16] M.A. Botchev, *expmARPACK: matrix exponential actions with Arnoldi methods*. Octave / Matlab software package, version 1.0, 2012. www.math.utwente.nl/ botchevma/expm/.

[17] M.A. Botchev, V. Grimm, M. Hochbruck, *Residual, restarting and Richardson iteration for the matrix exponential*, SIAM J. Sci. Comput. 35 (3) (2013) A1376A1397

[18] H.A. van der Vorst, *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems*, SIAM Journal on Scientific and Statistical Computing, 1992, Vol. 13, No. 2 : pp. 631-644, http://dx.doi.org/10.1137/0913035

[19] L.T. Yang, R.P. Brent, *The Improved BiCGStab Method for Large and Sparse Unsymmetric Linear Systems on Parallel Distributed Memory Architectures*

A PML derivation

As noted in Section 2.1, the *stretched coordinate* PML 6 can be derived by applying the following three transformations to the Maxwell equations [6]:

$$\partial_x \to \left(1 + i\frac{\sigma_x}{\omega}\right)^{-1} \partial_x,$$
$$\partial_y \to \left(1 + i\frac{\sigma_y}{\omega}\right)^{-1} \partial_y,$$
$$\partial_z \to \left(1 + i\frac{\sigma_z}{\omega}\right)^{-1} \partial_z.$$

Below is the system that is obtained by applying those transformations:

$$\begin{aligned}
\partial_t H_x &= -i\omega H_x = -\mu^{-1}(-(1+i\sigma_z/\omega)^{-1}\partial_z E_y + (1+i\sigma_y/\omega)^{-1}\partial_y E_z),\\
\partial_t H_y &= -i\omega H_y = -\mu^{-1}(\ \ (1+i\sigma_z/\omega)^{-1}\partial_z E_x - (1+i\sigma_x/\omega)^{-1}\partial_x E_z),\\
\partial_t H_z &= -i\omega H_z = -\mu^{-1}(-(1+i\sigma_y/\omega)^{-1}\partial_y E_x + (1+i\sigma_x/\omega)^{-1}\partial_x E_y),\\
\partial_t E_x &= -i\omega E_x = \ \ \epsilon^{-1}(-(1+i\sigma_z/\omega)^{-1}\partial_z H_y + (1+i\sigma_y/\omega)^{-1}\partial_y H_z),\\
\partial_t E_y &= -i\omega E_y = \ \ \epsilon^{-1}(\ \ (1+i\sigma_z/\omega)^{-1}\partial_z H_x - (1+i\sigma_x/\omega)^{-1}\partial_x H_z),\\
\partial_t E_z &= -i\omega E_z = \ \ \epsilon^{-1}(-(1+i\sigma_y/\omega)^{-1}\partial_y H_x + (1+i\sigma_x/\omega)^{-1}\partial_x H_y).
\end{aligned}$$

Multiplying left and right by the reciprocal of the transformations, we obtain:

$$\begin{aligned}
-(1+i\sigma_y/\omega)(1+i\sigma_z/\omega)i\omega H_x &= -\mu^{-1}(-(1+i\sigma_y/\omega)\partial_z E_y + (1+i\sigma_z/\omega)\partial_y E_z),\\
-(1+i\sigma_x/\omega)(1+i\sigma_z/\omega)i\omega H_y &= -\mu^{-1}(\ \ (1+i\sigma_x/\omega)\partial_z E_x - (1+i\sigma_z/\omega)\partial_x E_z),\\
-(1+i\sigma_x/\omega)(1+i\sigma_y/\omega)i\omega H_z &= -\mu^{-1}(-(1+i\sigma_x/\omega)\partial_y E_x + (1+i\sigma_y/\omega)\partial_x E_y),\\
-(1+i\sigma_y/\omega)(1+i\sigma_z/\omega)i\omega E_x &= \ \ \epsilon^{-1}(-(1+i\sigma_y/\omega)\partial_z H_y + (1+i\sigma_z/\omega)\partial_y H_z),\\
-(1+i\sigma_x/\omega)(1+i\sigma_z/\omega)i\omega E_y &= \ \ \epsilon^{-1}(\ \ (1+i\sigma_x/\omega)\partial_z H_x - (1+i\sigma_z/\omega)\partial_x H_z),\\
-(1+i\sigma_x/\omega)(1+i\sigma_y/\omega)i\omega E_z &= \ \ \epsilon^{-1}(-(1+i\sigma_x/\omega)\partial_y H_x + (1+i\sigma_y/\omega)\partial_x H_y).
\end{aligned}$$

Writing out the factors on the left hand side of these equations and taking the reciprocal of the transformation inside the differential operator on the right hand side result in:

$$\begin{aligned}
-i\omega H_x + (\sigma_y+\sigma_z)H_x + (i/\omega)\sigma_y\sigma_z H_x &= -\mu^{-1}(-\partial_z(E_y + (i/\omega)\sigma_y E_y) + \partial_y(E_z + (i/\omega)\sigma_z E_z)),\\
-i\omega H_y + (\sigma_x+\sigma_z)H_y + (i/\omega)\sigma_x\sigma_z H_y &= -\mu^{-1}(\ \ \partial_z(E_x + (i/\omega)\sigma_x E_x) - \partial_x(E_z + (i/\omega)\sigma_z E_z)),\\
-i\omega H_z + (\sigma_x+\sigma_y)H_z + (i/\omega)\sigma_x\sigma_y H_z &= -\mu^{-1}(-\partial_y(E_x + (i/\omega)\sigma_x E_x) + \partial_x(E_y + (i/\omega)\sigma_y E_y)),\\
-i\omega E_x + (\sigma_y+\sigma_z)E_x + (i/\omega)\sigma_y\sigma_z E_x &= \ \ \epsilon^{-1}(-\partial_z(H_y + (i/\omega)\sigma_y H_y) + \partial_y(H_z + (i/\omega)\sigma_z H_z)),\\
-i\omega E_y + (\sigma_x+\sigma_z)E_y + (i/\omega)\sigma_x\sigma_z E_y &= \ \ \epsilon^{-1}(\ \ \partial_z(H_x + (i/\omega)\sigma_x H_x) - \partial_x(H_z + (i/\omega)\sigma_z H_z)),\\
-i\omega E_z + (\sigma_x+\sigma_y)E_z + (i/\omega)\sigma_x\sigma_y E_z &= \ \ \epsilon^{-1}(-\partial_y(H_x + (i/\omega)\sigma_x H_x) + \partial_x(H_y + (i/\omega)\sigma_y H_y)).
\end{aligned}$$

These relations can then be written in vector form as:

$$\begin{aligned}
\partial_t \mathbf{H} &= -\mu^{-1}(\nabla\times(\mathbf{E}+\mathbf{P})) - \begin{pmatrix} \sigma_y+\sigma_z & 0 & 0\\ 0 & \sigma_x+\sigma_z & 0\\ 0 & 0 & \sigma_x+\sigma_y \end{pmatrix}\mathbf{H} + \mathbf{R},\\
\partial_t \mathbf{E} &= \ \ \epsilon^{-1}(\nabla\times(\mathbf{H}+\mathbf{Q})) - \begin{pmatrix} \sigma_y+\sigma_z & 0 & 0\\ 0 & \sigma_x+\sigma_z & 0\\ 0 & 0 & \sigma_x+\sigma_y \end{pmatrix}\mathbf{E} + \mathbf{S},\\
\mathbf{P} &= (i/\omega)\mathrm{diag}(\sigma)\mathbf{E} \Leftrightarrow -i\omega\mathbf{P} = \partial_t\mathbf{P} = \mathrm{diag}(\sigma)\mathbf{E},\\
\mathbf{Q} &= (i/\omega)\mathrm{diag}(\sigma)\mathbf{H} \Leftrightarrow -i\omega\mathbf{Q} = \partial_t\mathbf{Q} = \mathrm{diag}(\sigma)\mathbf{H},\\
\mathbf{R} &= (i/\omega)\begin{pmatrix} \sigma_y\sigma_z & 0 & 0\\ 0 & \sigma_x\sigma_z & 0\\ 0 & 0 & \sigma_x\sigma_y \end{pmatrix}\mathbf{H} \Leftrightarrow -i\omega\mathbf{R} = \partial_t\mathbf{R} = \begin{pmatrix} \sigma_y\sigma_z & 0 & 0\\ 0 & \sigma_x\sigma_z & 0\\ 0 & 0 & \sigma_x\sigma_y \end{pmatrix}\mathbf{H},\\
\mathbf{S} &= (i/\omega)\begin{pmatrix} \sigma_y\sigma_z & 0 & 0\\ 0 & \sigma_x\sigma_z & 0\\ 0 & 0 & \sigma_x\sigma_y \end{pmatrix}\mathbf{E} \Leftrightarrow -i\omega\mathbf{S} = \partial_t\mathbf{S} = \begin{pmatrix} \sigma_y\sigma_z & 0 & 0\\ 0 & \sigma_x\sigma_z & 0\\ 0 & 0 & \sigma_x\sigma_y \end{pmatrix}\mathbf{E}.
\end{aligned}$$

B Matlab Codes

The Matlab code computes the Maxwell operator and the preconditioner matrices.

```
function [A,A1,A2] = Maxwell_blocks3D(nx,ny,nz,pars,precond_type)

% The cuboid domain [ax,bx]x[ay,by]x[az,bz] is divided into nx x ny x nz cells
%
% The whole domain [ax,bx] x [ay,by] x [az,bz]  includes the PML regions.
% The boundary conditions are homogeneous Dirichlet (outside of the PML
% regions)
%
% pars.feval_mesh     handle for
%                     [x,y,z,hx,hy,hz] = feval(pars.feval_mesh,nx,ny,nz);
% pars.feval_domain   handle for
%                     [ax,bx,ay,by,az,bz] = feval(pars.feval_domain,ok_pml);
% pars.feval_sigma    handle for
%                     sigmaXYZ = feval(pars.feval_sigma,X,Y,Z,what)
%                     where "what" can be "Ex", "Hx", "Hy", ...
% pars.feval_mu       handle for
%                     muXYZ = feval(pars.feval_mu,X,Y,Z)
% pars.what           'no_pml' or '2D' or '3D'.  If pars.what=='no_pml'
%                     then only the Maxwell operator itself is computed.
%                     Otherwise the matrix is computed, including the PML
%                     parts in x and y directions for 2D and x,y,z for 3D.

[x,y,z,hx,hy,hz] = feval(pars.feval_mesh, nx, ny, nz);

[X,Y,Z] = ndgrid(x, y, z);
N = (nx + 1)*(ny + 1)*(nz + 1);

% Defining block top right
Dx=derivative1D(nx+1, hx);
Dy=derivative1D(ny+1, hy);
Dz=derivative1D(nz+1, hz);
Ix=speye(nx+1, nx+1);
Iy=speye(ny+1, ny+1);
Iz=speye(nz+1, nz+1);

dEy_dz = kron3(Dz, Iy, Ix); % dEy_dz, etc, componenten dubbel
dEz_dy = kron3(Iz, Dy, Ix);
dEx_dz = kron3(Dz, Iy, Ix);
dEz_dx = kron3(Iz, Iy, Dx);
dEx_dy = kron3(Iz, Dy, Ix);
dEy_dx = kron3(Iz, Iy, Dx);

mu_x = feval(pars.feval_mu, X, Y+hy/2, Z+hz/2); % the mu values for Hx
mu_y = feval(pars.feval_mu, X+hx/2, Y, Z+hz/2); % the mu values for Hy
mu_z = feval(pars.feval_mu, X+hx/2, Y+hy/2, Z); % the mu values for Hz
Mu_x_inv = spdiags(1./mu_x(:), 0, N, N);
Mu_y_inv = spdiags(1./mu_y(:), 0, N, N);
Mu_z_inv = spdiags(1./mu_z(:), 0, N, N);

B15 =  Mu_x_inv*dEy_dz;
B16 = -Mu_x_inv*dEz_dy;
B24 = -Mu_y_inv*dEx_dz;
B26 =  Mu_y_inv*dEz_dx;
B34 =  Mu_z_inv*dEx_dy;
B35 = -Mu_z_inv*dEy_dx;

% Defining block bottom left
epsilon_x = feval(pars.feval_eps, X+hx/2, Y, Z); % the epsilon values for Ex
epsilon_y = feval(pars.feval_eps, X, Y+hy/2, Z); % the epsilon values for Ey
epsilon_z = feval(pars.feval_eps, X, Y, Z+hz/2); % the epsilon values for Ez
Eps_x_inv = spdiags(1./epsilon_x(:), 0, N, N);
Eps_y_inv = spdiags(1./epsilon_y(:), 0, N, N);
Eps_z_inv = spdiags(1./epsilon_z(:), 0, N, N);

```

```
B42 =  Eps_x_inv*dEy_dz';
B43 = -Eps_x_inv*dEz_dy';
B51 = -Eps_y_inv*dEx_dz';
B53 =  Eps_y_inv*dEz_dx';
B61 =  Eps_z_inv*dEx_dy';
B62 = -Eps_z_inv*dEy_dx';

%define diagonal blocks for H
sigma_x = feval(pars.feval_sigma, X+hx/2, 'x');
sigma_y = feval(pars.feval_sigma, Y+hy/2, 'y');
sigma_z = feval(pars.feval_sigma, Z+hz/2, 'z');
Sigma_x = spdiags(sigma_x(:), 0, N, N);
Sigma_y = spdiags(sigma_y(:), 0, N, N);
Sigma_z = spdiags(sigma_z(:), 0, N, N);

B11 = - Sigma_y - Sigma_z;
B22 = - Sigma_x - Sigma_z;
B33 = - Sigma_x - Sigma_y;

%define diagonal blocks for E
sigma_x = feval(pars.feval_sigma, X, 'x');
sigma_y = feval(pars.feval_sigma, Y, 'y');
sigma_z = feval(pars.feval_sigma, Z, 'z');
Sigma_x = spdiags(sigma_x(:), 0, N, N);
Sigma_y = spdiags(sigma_y(:), 0, N, N);
Sigma_z = spdiags(sigma_z(:), 0, N, N);

B44 = - Sigma_y - Sigma_z;
B55 = - Sigma_x - Sigma_z;
B66 = - Sigma_x - Sigma_y;

% Whole matrix
O = sparse(N,N); % all zero sparse matrix

A=[ B11,   O,   O,   O, B15, B16; ...
      O, B22,   O, B24,   O, B26; ...
      O,   O, B33, B34, B35,   O; ...
      O, B42, B43, B44,   O,   O; ...
    B51,   O, B53,   O, B55,   O; ...
    B61, B62,   O,   O,   O, B66];

switch precond_type
    case 0 %no preconditioning
        A1 = speye(size(A));
        A2 = A1;
    case 1 %ADI splitting
        A1=[ B11,   O,   O,   O,   O, B16; ...
               O, B22,   O, B24,   O,   O; ...
               O,   O, B33,   O, B35,   O; ...
               O, B42,   O, B44,   O,   O; ...
               O,   O, B53,   O, B55,   O; ...
             B61,   O,   O,   O,   O, B66];

        A2=[ B11,   O,   O,   O, B15,   O; ...
               O, B22,   O,   O,   O, B26; ...
               O,   O, B33, B34,   O,   O; ...
               O,   O, B43, B44,   O,   O; ...
             B51,   O,   O,   O, B55,   O; ...
               O, B62,   O,   O,   O, B66];
    case 2 %field splitting
        A1=[ B11,   O,   O,   O, B15, B16; ...
               O, B22,   O, B24,   O, B26; ...
               O,   O, B33, B34, B35,   O; ...
               O,   O,   O,   O,   O,   O; ...
               O,   O,   O,   O,   O,   O; ...
               O,   O,   O,   O,   O,   O];

        A2=[   O,   O,   O,   O,   O,   O; ...
               O,   O,   O,   O,   O,   O; ...
```

```
134                 O,   O,   O,   O,   O,   O; ...
135                 O, B42, B43, B44,   O,   O; ...
136               B51,   O, B53,   O, B55,   O; ...
137               B61, B62,   O,   O,   O, B66 ];
138 end
139
140 if strcmp(pars.what,'no_pml')
141    return
142 elseif strcmp(pars.what,'2D')
143     % Add PML 2D
144     switch precond_type
145         case 0
146             I = speye(N);
147
148             sigma_x1 = feval(pars.feval_sigma, X, 'x');
149             sigma_y1 = feval(pars.feval_sigma, Y, 'y');
150             Sigma_x1 = spdiags(sigma_x1(:), 0, N, N);
151             Sigma_y1 = spdiags(sigma_y1(:), 0, N, N);
152             nonzero_sig_x1 = sigma_x1>0;
153             nonzero_sig_y1 = sigma_y1>0;
154             nonzero_sig_xy1= (sigma_x1.*sigma_y1)>0;
155
156             sigma_x2 = feval(pars.feval_sigma, X+hx/2, 'x');
157             sigma_y2 = feval(pars.feval_sigma, Y+hy/2, 'y');
158             Sigma_x2 = spdiags(sigma_x2(:), 0, N, N);
159             Sigma_y2 = spdiags(sigma_y2(:), 0, N, N);
160             nonzero_sig_x2 = sigma_x2>0;
161             nonzero_sig_y2 = sigma_y2>0;
162             nonzero_sig_xy2= (sigma_x2.*sigma_y2)>0;
163
164             A = [A, [O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                     O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), ...
                     O(:,nonzero_sig_xy2); ...
165                      O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                             O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                             O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
166                      O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                             O(:,nonzero_sig_xy1), I(:,nonzero_sig_x2), ...
                             I(:,nonzero_sig_y2), I(:,nonzero_sig_xy2); ...
167                      O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                             O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                             O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
168                      O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                             O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                             O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
169                      I(:,nonzero_sig_x1), I(:,nonzero_sig_y1), ...
                             I(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                             O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)]];
170
171             B71  = Sigma_x1*B61;
172             B82  = Sigma_y1*B62;
173             B96  = -Sigma_x1*Sigma_y1;
174             B104 = Sigma_x2*B34;
175             B115 = Sigma_y2*B35;
176             B123 = -Sigma_x2*Sigma_y2;
177
178             Block_row_7  = [B71,   O,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
179             Block_row_8  = [  O, B82,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
180             Block_row_9  = [  O,   O,    O,    O,    O, B96,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
181             Block_row_10 = [  O,   O,    O, B104,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
```

```
182             Block_row_11 = [  O,   O,    O,    O, B115,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
183             Block_row_12 = [  O,   O, B123,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
184
185             A = [A; ...
186                 Block_row_7(nonzero_sig_x1,:); ...
187                 Block_row_8(nonzero_sig_y1,:); ...
188                 Block_row_9(nonzero_sig_xy1,:); ...
189                 Block_row_10(nonzero_sig_x2,:); ...
190                 Block_row_11(nonzero_sig_y2,:); ...
191                 Block_row_12(nonzero_sig_xy2,:)];
192         case 1
193             I = speye(N);
194
195             sigma_x1 = feval(pars.feval_sigma, X, 'x');
196             sigma_y1 = feval(pars.feval_sigma, Y, 'y');
197             Sigma_x1 = spdiags(sigma_x1(:), 0, N, N);
198             Sigma_y1 = spdiags(sigma_y1(:), 0, N, N);
199             nonzero_sig_x1 = sigma_x1>0;
200             nonzero_sig_y1 = sigma_y1>0;
201             nonzero_sig_xy1= (sigma_x1.*sigma_y1)>0;
202
203             sigma_x2 = feval(pars.feval_sigma, X+hx/2, 'x');
204             sigma_y2 = feval(pars.feval_sigma, Y+hy/2, 'y');
205             Sigma_x2 = spdiags(sigma_x2(:), 0, N, N);
206             Sigma_y2 = spdiags(sigma_y2(:), 0, N, N);
207             nonzero_sig_x2 = sigma_x2>0;
208             nonzero_sig_y2 = sigma_y2>0;
209             nonzero_sig_xy2= (sigma_x2.*sigma_y2)>0;
210
211             B71  = Sigma_x1*B61;
212             B82  = Sigma_y1*B62;
213             B96  = -Sigma_x1*Sigma_y1;
214             B104 = Sigma_x2*B34;
215             B115 = Sigma_y2*B35;
216             B123 = -Sigma_x2*Sigma_y2;
217
218             Block_row_7  = [B71,   O,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
219             Block_row_8  = [  O, B82,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
220             Block_row_9  = [  O,   O,    O,    O,    O, B96,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
221             Block_row_10 = [  O,   O,    O, B104,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
222             Block_row_11 = [  O,   O,    O,    O, B115,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
223             Block_row_12 = [  O,   O, B123,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
224             Block_row_zeros =[O,   O,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
225
226             A1 = [A1, [O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                    O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), ...
                    O(:,nonzero_sig_xy2); ...
227                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
```

```
228                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
229                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
230                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
231                     I(:,nonzero_sig_x1), I(:,nonzero_sig_y1), ...
                            I(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)]];
232
233             A1 = [A1; ...
234                   Block_row_7(nonzero_sig_x1,:); ...
235                   Block_row_8(nonzero_sig_y1,:); ...
236                   Block_row_9(nonzero_sig_xy1,:); ...
237                   Block_row_zeros(nonzero_sig_x2,:); ...
238                   Block_row_zeros(nonzero_sig_y2,:); ...
239                   Block_row_zeros(nonzero_sig_xy2,:)];
240
241             A2 = [A2, [O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                        O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), ...
                        O(:,nonzero_sig_xy2); ...
242                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
243                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), I(:,nonzero_sig_x2), ...
                            I(:,nonzero_sig_y2), I(:,nonzero_sig_xy2); ...
244                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
245                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
246                     O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                            O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)]];
247
248             A2 = [A2; ...
249                   Block_row_zeros(nonzero_sig_x1,:); ...
250                   Block_row_zeros(nonzero_sig_y1,:); ...
251                   Block_row_zeros(nonzero_sig_xy1,:); ...
252                   Block_row_10(nonzero_sig_x2,:); ...
253                   Block_row_11(nonzero_sig_y2,:); ...
254                   Block_row_12(nonzero_sig_xy2,:)];
255         case 2
256             I = speye(N);
257
258             sigma_x1 = feval(pars.feval_sigma, X, 'x');
259             sigma_y1 = feval(pars.feval_sigma, Y, 'y');
260             Sigma_x1 = spdiags(sigma_x1(:), 0, N, N);
261             Sigma_y1 = spdiags(sigma_y1(:), 0, N, N);
262             nonzero_sig_x1 = sigma_x1>0;
263             nonzero_sig_y1 = sigma_y1>0;
264             nonzero_sig_xy1= (sigma_x1.*sigma_y1)>0;
265
266             sigma_x2 = feval(pars.feval_sigma, X+hx/2, 'x');
267             sigma_y2 = feval(pars.feval_sigma, Y+hy/2, 'y');
268             Sigma_x2 = spdiags(sigma_x2(:), 0, N, N);
269             Sigma_y2 = spdiags(sigma_y2(:), 0, N, N);
270             nonzero_sig_x2 = sigma_x2>0;
271             nonzero_sig_y2 = sigma_y2>0;
272             nonzero_sig_xy2= (sigma_x2.*sigma_y2)>0;
273
274             B71  = Sigma_x1*B61;
275             B82  = Sigma_y1*B62;
276             B96  = -Sigma_x1*Sigma_y1;
```

```
277             B104 = Sigma_x2*B34;
278             B115 = Sigma_y2*B35;
279             B123 = -Sigma_x2*Sigma_y2;
280
281             Block_row_7  = [B71,   O,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
282             Block_row_8  = [  O, B82,    O,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
283             Block_row_9  = [  O,   O,    O,    O,    O, B96,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
284             Block_row_10 = [  O,   O,    O, B104,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
285             Block_row_11 = [  O,   O,    O,    O, B115,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
286             Block_row_12 = [  O,   O, B123,    O,    O,   O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
287             Block_row_zeros =[O,   O,    O,    O,    O,   ...
                    O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), O(:,nonzero_sig_xy1), ...
                    O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)];
288
289             A1 = [A1, [O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                    O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), ...
                    O(:,nonzero_sig_xy2); ...
290                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
291                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), I(:,nonzero_sig_x2), ...
                              I(:,nonzero_sig_y2), I(:,nonzero_sig_xy2); ...
292                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
293                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
294                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)]];
295
296             A1 = [A1; ...
297                 Block_row_zeros(nonzero_sig_x1,:); ...
298                 Block_row_zeros(nonzero_sig_y1,:); ...
299                 Block_row_zeros(nonzero_sig_xy1,:); ...
300                 Block_row_10(nonzero_sig_x2,:); ...
301                 Block_row_11(nonzero_sig_y2,:); ...
302                 Block_row_12(nonzero_sig_xy2,:)];
303
304             A2 = [A2, [O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                    O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), O(:,nonzero_sig_y2), ...
                    O(:,nonzero_sig_xy2); ...
305                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
306                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
307                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
308                       O(:,nonzero_sig_x1), O(:,nonzero_sig_y1), ...
                              O(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                              O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2); ...
```

```
                        I(:,nonzero_sig_x1), I(:,nonzero_sig_y1), ...
                            I(:,nonzero_sig_xy1), O(:,nonzero_sig_x2), ...
                            O(:,nonzero_sig_y2), O(:,nonzero_sig_xy2)]];

            A2 = [A2; ...
                Block_row_7(nonzero_sig_x1,:); ...
                Block_row_8(nonzero_sig_y1,:); ...
                Block_row_9(nonzero_sig_xy1,:); ...
                Block_row_zeros(nonzero_sig_x2,:); ...
                Block_row_zeros(nonzero_sig_y2,:); ...
                Block_row_zeros(nonzero_sig_xy2,:)];
    end
else
    % Add PML 3D
    I = speye(N);
    A = [A, [O,   O,   O,   O, B15, B16, I, O, O, O, O, O; ...
             O,   O,   O, B24,   O, B26, O, I, O, O, O, O; ...
             O,   O,   O, B34, B35,   O, O, O, I, O, O, O; ...
             O, B42, B43,   O,   O,   O, O, O, O, I, O, O; ...
           B51,   O, B53,   O,   O,   O, O, O, O, O, I, O; ...
           B61, B62,   O,   O,   O,   O, O, O, O, O, O, I]];

    sigma_x = feval(pars.feval_sigma, X, 'x');
    sigma_y = feval(pars.feval_sigma, Y, 'y');
    sigma_z = feval(pars.feval_sigma, Z, 'z');
    Sigma_x = spdiags(sigma_x(:), 0, N, N);
    Sigma_y = spdiags(sigma_y(:), 0, N, N);
    Sigma_z = spdiags(sigma_z(:), 0, N, N);
    B71  = Sigma_x;
    B82  = Sigma_y;
    B93  = Sigma_z;
    BEx = -Sigma_y*Sigma_z;
    BEy = -Sigma_x*Sigma_z;
    BEz = -Sigma_x*Sigma_y;

    sigma_x = feval(pars.feval_sigma, X+hx/2, 'x');
    sigma_y = feval(pars.feval_sigma, Y+hy/2, 'y');
    sigma_z = feval(pars.feval_sigma, Z+hz/2, 'z');
    Sigma_x = spdiags(sigma_x(:), 0, N, N);
    Sigma_y = spdiags(sigma_y(:), 0, N, N);
    Sigma_z = spdiags(sigma_z(:), 0, N, N);
    B104 = Sigma_x;
    B115 = Sigma_y;
    B126 = Sigma_z;
    BHx = -Sigma_y*Sigma_z;
    BHy = -Sigma_x*Sigma_z;
    BHz = -Sigma_x*Sigma_y;

    A = [A; [   O    O    O B71   O   O O O O O O O O O O O; ...
                O    O    O   O B82   O O O O O O O O O O O; ...
                O    O    O   O   O B93 O O O O O O O O O O; ...
             B104    O    O   O   O   O O O O O O O O O O O; ...
                O B115    O   O   O   O O O O O O O O O O O; ...
                O    O B126   O   O   O O O O O O O O O O; ...
              BHx    O    O   O   O   O O O O O O O O O O; ...
                O  BHy    O   O   O   O O O O O O O O O O; ...
                O    O  BHz   O   O   O O O O O O O O O O; ...
                O    O    O BEx   O   O O O O O O O O O O; ...
                O    O    O   O BEy   O O O O O O O O O O; ...
                O    O    O   O   O BEz O O O O O O O O O O] ];

end
% ---------------------------------------------------------
function D_xyz = derivative1D(n_xyz_p1, h_xyz)
% computes 1D differential operator of vector component of E in any direction
ee   = ones(n_xyz_p1, 1)/h_xyz;
D_xyz = spdiags([-ee,ee], [0 1], n_xyz_p1, n_xyz_p1);

% ---------------------------------------------------------
```

```
376 function D = kron3(A, B, C)
377 D=kron(A, kron(B, C));
```