Finite-Difference Time Domain Method (FDTD)

Course: Computational Photonics

Group number - 2

Group members - Akshara Yagnik, Pooja Asok Kumar, Shiyue Fan, Xiong Ziyi

Introdcution to the Finite-Difference Time Domain Method (FDTD):

The FDTD method is a rigorous, numerical technique to simulate evolution of electric and magnetic fields in the time domain using the Maxwell's equations. In its 1D form, the method is actually implemented for one spacial dimension (here we consider in x-direction) and one temporal dimension. So, its a (1 + 1D) problem. Here we assume no dependance on y and z direction. So the spatio-temporal dynamics (in x and t) of the y-polarized magnetic field component and the z-polarized electric field and component and is governed by the following equations with j_z being the current source:

(i) $\frac{\partial H_y(x,t)}{\partial x} = \epsilon_0 \epsilon_r(x) \frac{\partial E_z(x,t)}{\partial t} + j_z(x,t)$

(ii) $\frac{\partial E_z(x,t)}{\partial x} = \mu_0 \frac{\partial H_y(x,t)}{\partial t}$

Space and time are discretized respectively as $x = i\Delta x$, and t = nΔt. The time index (upper index) is n, and space index (lower index) is i. We follow the below format to then approximate the partial differential equations (i) and (ii) using the finite difference scheme :

(iii) $E_z(x,t) \Rightarrow E_z(i\Delta x, n\Delta t) = E_i^n$

(iv) $H_y(x,t) \Rightarrow H_y(i\Delta x, n\Delta t) = H_i^n$

(v) $\epsilon(x) \Rightarrow \epsilon(i\Delta x) = \epsilon_i$

As we move on to the finite difference approximation of Maxwell's eqautions, note that the principle of causality is integral to this scheme since fields in the past determine fields in the future:

(vi) $E_{z,i}^{n+1} = E_{z,i}^{n} + \frac{\Delta t}{\epsilon_0 \epsilon_{r,i}} (\frac{H_{y,i+0.5}^{n+0.5} - H_{y,i-0.5}^{n+0.5}}{\Delta x} - j_{z,i}^{n+0.5})$

(vii) $H_{y,i+0.5}^{n+1.5} = H_{y,i+0.5}^{n+0.5} + \frac{\Delta t}{\mu_0 \Delta x} \frac{E_{z,i+1}^{n+1} - E_{z,i}^{n+1}}{\Delta x}$

Yee grid in 1D and Leapfrog time steps:

Since equations (vi) and (vii) are coupled to each other, it seems as f the information on specific feild in time is always skipping over the nearest future field. In other words, it influences only the next to the nearest field. Yee's grid offers a spatial and temporal discretization that is different for H_y and E_z. The difference of half a discretization step helps in decoupling of the two equations giving us the following results:

(viii) $E_i^{n+1} = E_i^n + \frac{1}{\epsilon_o \epsilon_i} \frac{\Delta t}{\Delta x} [H_{i+0.5}^{n+0.5} - H_{i-0.5}^{n+0.5}]$

(ix) $H_{i+0.5}^{n+1.5} = H_{i+0.5}^{n+0.5} + \frac{1}{\mu_0} \frac{\Delta t}{\Delta x} [E_{i+1}^{n+1} - H_i^{n+1}]$

These decoupled equations allow interative calculation of electromagnetic fields in the future using knowledge of the present electromagnetic fields. This is done using leapfrog time-stepping as described in words below. (Note that this scheme is a fully explicit iteration.)

- Using knowledge of present distribution of the magnetic field $H_{i+0.5}^{n+0.5}$ and past distribution of he electric field E_i^n , future spatial distribution of the electric field E_i^{n+1} is calculated.
- Following which using the previously calculated E_i^{n+1}, and present distribution of $H_{i+0.5}^{n+0.5}$, the future spatial distribution of magetic field $H_{i+0.5}^{n+1.5}$ is calculated.
- Until the fields are known for the entire time interval of interest, the above described 2-step procedure is repeated.

Objective the code below:

In this homeowork, we start by implementing the Finite Difference Time Domain method (FDTD) in its 1D and 3D versions. Their implementation is tested by first simulating propagation of an ultrashort pulse in a homogeneous medium. This is done in a dispersion-free dielectric medium $\epsilon(x) = 1$. Then we introduce an interface located at a distance 4.5 μm in positive direction from the centre of the computational domain. This interface lies between two different dielectric media with permitivities $\epsilon_1 = 1$ and $\epsilon_1 = 4$, so we next simulate propagation of the ultrashort pulse through an inhomogeneous medium.

Task 1: Implement the FDTD method in 1D and 3D versions

1.1 Implementing the FDTD method in 1D

Code explanation:

Let's go through the provided Python code step by step to understand what each part is doing.

Constants

• Speed of Light (c): The constant c represents the speed of light in a vacuum.

• Vacuum Permeability (mu0): The constant mu0 is the permeability of free space.

• Vacuum Permittivity (epsilon_0): It represents the permittivity of free space, which is a measure of how an electric field affects, and is affected by, a vacuum.

• Vacuum Impedance (Z0): representing the characteristic impedance of free space.

Geometry Parameters • Computational Domain Width (x_width): The width of the computational domain is set to 18 micrometers.

• Refractive Indices (n1 and n2): The refractive index n1 is set to 1 and n2 as 2 (which could represent refractive index).

• Interface Position (interf_pos): The position of the dielectric interface is set to a quarter of the computational domain width, which is 4.5 micro meters.

Simulation Parameters

• Grid Spacing (dx): The spatial resolution of the simulation grid is set to 15 nanometers.

• Time Step (dt): The duration of each simulation time step is set to 60 femtoseconds.

• Number of Grid Points (Nx): The total number of grid points along the x-axis is calculated by dividing the computational domain width by the grid spacing and adding 1. This is rounded to the nearest integer using the round function.

Source Parameters

• Source Frequency (source_frequency): The frequency of the source is set to 500 Terahertz.

• Source Position (source_position): The position of the source is set to 0 meters, meaning it is located at the origin of the computational domain.

• Source Pulse Length (source_pulse_length): The duration of the source pulse is set to 1 femtosecond. The function fdtd_1d computes the temporal evolution of a pulsed excitation using the 1D Finite-Difference Time-Domain (FDTD) method. It returns the electric field Ez and the magnetic field Hy as 2D arrays, along with the spatial and temporal coordinates.

Constants and Initializations

• del_t is the time step for the simulation, calculated based on the Courant condition for stability.

Plot for Dielectric Interface

• A similar plot is created for Sx, the Poynting vector with the dielectric interface.

• The same normalization (vmax) is used to allow direct comparison between the two scenarios.

• The plot is displayed using plt.show(), and saved as Sx_1D.pdf if save_figures is True.

• e_factor and h_factor are prefactors used in the update equations for the electric and magnetic fields.

• Nx is the number of spatial grid points, determined by the size of eps_rel.

• x is the array of spatial coordinates, centered around zero.

• Numb_iter is the number of time steps for the simulation.

• t is the array of time steps.

Field Arrays

Ez and Hy are arrays to store the electric and magnetic field values at each time step and spatial position. They are initialized to zero.

Source Properties

• source_angular_frequency is the angular frequency of the source.

• t0 is the time offset for the center of the Gaussian pulse.

• source_ind is the index of the source position on the grid. A check ensures the source is within valid bounds.

Main FDTD Loop

• The loop iterates over each time step. • The electric field Ez is updated using the finite difference approximation of Maxwell's curl equations. • The source term is added to the electric field at the specified position and time.

The source is given as: j_source= (exp(-1j * source_(angular_frequency)* t_source)*exp(-(t_source/ source_(pulse_length))* 2)) Where exp(-1j * source_angular_frequency * t_source) is the carrier and exp(-(t_source / source_pulse_length) ** 2) is the envelop.

• The magnetic field Hy is updated similarly. • The loop iterates over each time step. • The electric field Ez is updated using the finite difference approximation of Maxwell's curl equations. • The source term is added to the electric field at the specified position and time. • The magnetic field Hy is updated similarly.

Post-Processing

• Hy is interpolated to match the same spatial and temporal grid as Ez.

• average_axes and replicate_boundary_values functions are used to handle boundary conditions and averaging, ensuring consistent field values.

• The function returns the electric and magnetic fields, along with the spatial and temporal coordinates.

The avg_axes function computes the average of neighboring values along the specified axes of a multi-dimensional array (field).

Prefactor for Averaging

The variable f is a prefactor set to 0.5, ensuring that the averages are calculated with the same numerical precision as the input field. This helps in avoiding precision loss during computation.

Slice Definitions

• c represents the full slice of the axis, equivalent to [:].

• l (left) is a slice from the start to the second-to-last element.

• r (right) is a slice from the second element to the end.

Main Averaging Loop

• res is initialized to the input field.

• The loop iterates over each specified axis in axes.

• For each axis ax, tuples a and b are created to represent the left and right slices along that axis, respectively. For dimensions not equal to ax, the full slice c is used.

• res[a] and res[b] extract the neighboring elements along the axis ax.

• res is updated by taking the average of res[a] and res[b], weighted by the prefactor f.

• After processing all specified axes, the function returns the averaged field.

The repl_bound_val function pads the input field array by replicating its boundary values along the specified axes.

Main Replication Loop

• res is initialized to the input field.

• The loop iterates over each specified axis in axes.

For each axis ax:

1. Creating Slices for Boundary Replication:

o a: A tuple representing the slice to extract the first element along axis ax. For all other axes, it represents a full slice (slice(None)).

o b: A tuple representing the slice to extract the last element along axis ax. For all other axes, it represents a full slice (slice(None)).

o na: A tuple representing the insertion of a new axis (np.newaxis) at the position of ax, and full slices for other axes. This is used to match the dimensions when concatenating.

2 .Extracting and Replicating Boundary Values:

• res[a][na] extracts the first element along axis ax and adds a new axis at ax.

• res[b][na] extracts the last element along axis ax and adds a new axis at ax.

• These extracted slices are then concatenated with the original res along axis ax, effectively replicating the boundary values.

3. Updating the Result:

• The updated res now includes the replicated boundary values for the current axis.

The Timer class is a simple utility to measure elapsed time between events, similar to a stopwatch with start (tic) and stop (toc) functions.

Initializer (init)

• The **init** method initializes the Timer object.

• self._tic stores the current time when the object is created using time.time(), which returns the current time in seconds since the epoch (a floating point number).

tic Method

The tic method updates self._tic to the current time. This method is used to mark the start time from which the elapsed time will be measured.

toc Method

• The toc method calculates and returns the elapsed time in seconds since the last call to tic.

• It does this by subtracting the stored self._tic time from the current time obtained using time.time(). ** A Timer object is created, and the current time is stored in self._tic.**

• The Timer class allows you to measure elapsed time between two events. • You can use the tic method to start or reset the timer and the toc method to get the elapsed time since the last tic.

This utility can be particularly useful for measuring the performance or execution time of code blocks in your programs. Simulating Homogeneous Medium

• eps_rel is initialized as a 1D array with size Nx, filled with the relative permittivity of the homogeneous medium (n1**2).

• The function fdtd_1d is called to simulate the electromagnetic wave propagation in this homogeneous medium.

• Ez_ref and Hy_ref store the electric and magnetic fields respectively for reference.

• x and t store the spatial and temporal coordinates.

Simulating Dielectric Interface

This line modifies eps_rel to simulate a dielectric interface. For positions x greater than or equal to interf_pos, the relative permittivity is set to n2**2.

Timing the Simulation

• The timer is started using timer.tic().

• The fdtd_1d function is called again to simulate the electromagnetic wave propagation in the medium with the dielectric interface.

• The elapsed time for this simulation is printed using timer.toc().

Plotting the Results

Plotting Electric Field in Homogeneous Medium

• extent defines the bounds for the plot in micrometers, scaling the time and spatial coordinates.

• vmax is set to the maximum value of the absolute electric field in either simulation, scaled to microvolts per meter (µV/m).

• plt.imshow visualizes the real part of Ez_ref, scaling it and setting the color limits.

• The plot is labeled, and if save_figures is True, the figure is saved as Ez_ref_1D.pdf.

Plotting Electric Field with Dielectric Interface

• Another plot is created for Ez, the electric field with the dielectric interface.

• Similar steps are followed as for the homogeneous medium plot, including labeling and saving the figure as Ez_1D.pdf.

This code performs FDTD simulations to study electromagnetic wave propagation in two scenarios:

1. A homogeneous medium with a constant refractive index.
2. A medium with a dielectric interface, where the refractive index changes at a certain position.

The results of these simulations are visualized as time-space plots of the electric field, highlighting how the field behaves in each scenario. The use of a timer helps measure the computational time required for the simulation.

Calculate the Poynting Vector

The Poynting vector represents the power flow (energy transfer) per unit area in an electromagnetic field. For a 1D FDTD simulation, the Poynting vector in the xxx-direction (Sx) can be calculated as: Sx=−0.5 Re(Ez⋅conj(Hy)) where:

• Ez is the electric field in the z-direction.

• Hy is the magnetic field in the y-direction.

• Sx_ref is the Poynting vector for the homogeneous medium.

• Sx is the Poynting vector for the medium with the dielectric interface.

Plot Power Flow

The power flow is visualized by plotting the normalized Poynting vector for both simulations.

Plot for Homogeneous Medium

• vmax is set to the maximum absolute value of Sx_ref.

• plt.imshow visualizes Sx_ref, normalized by vmax, with a color scale from -1 to 1.

• Labels and colorbar are added to the plot.

• If save_figures is True, the figure is saved as Sx_ref_1D.pdf.

Plot for Dielectric Interface

• A similar plot is created for Sx, the Poynting vector with the dielectric interface.

• The same normalization (vmax) is used to allow direct comparison between the two scenarios.

• The plot is displayed using plt.show(), and saved as Sx_1D.pdf if save_figures is True.

In [ ]:
```
from matplotlib.colors import ListedColormap
import matplotlib.cm

cm_type = "diverging"
```

```
cm_data = [[0.65451717,0.83663233,0.93192758],
           [0.64547209,0.82872207,0.9272869 ],
           [0.63647241,0.82085477,0.92263815],
           [0.62749358,0.81302908,0.91804454],
           [0.61853579,0.80524414,0.91350348],
           [0.60959256,0.79749894,0.90902942],
           [0.60066198,0.78979259,0.90462466],
           [0.59174902,0.78212435,0.90027415],
           [0.58284727,0.77449324,0.89599145],
           [0.57395788,0.76689845,0.89177104],
           [0.56507855,0.75933908,0.88761581],
           [0.5562096 ,0.75181433,0.88352222],
           [0.5473473 ,0.74432326,0.87949636],
           [0.5384938 ,0.73686511,0.87553003],
           [0.52964595,0.72943898,0.87162775],
           [0.52079914,0.72204388,0.86779724],
           [0.51195734,0.71467914,0.86402567],
           [0.50311772,0.70734385,0.86031635],
           [0.49427494,0.70003697,0.85667816],
           [0.48543257,0.69275784,0.85309899],
           [0.4765839 ,0.68550534,0.84959049],
           [0.46773004,0.67827871,0.84614596],
           [0.45887158,0.67107716,0.84275984],
           [0.44999945,0.66389941,0.83944828],
           [0.44111696,0.6567448 ,0.836199  ],
           [0.4322209 ,0.64961233,0.83301432],
           [0.42331893,0.64249774,0.82990127],
           [0.41451465,0.63538296,0.826805  ],
           [0.40580609,0.62826205,0.82376836],
           [0.39728167,0.62112445,0.82071807],
           [0.38894427,0.61396324,0.81770077],
           [0.38090162,0.60676184,0.81466481],
           [0.37331891,0.59949648,0.81152804],
           [0.36645963,0.59212638,0.80818699],
           [0.36091373,0.58458211,0.80425245],
           [0.3576747 ,0.57680321,0.79866681],
           [0.35640365,0.56899945,0.79048117],
           [0.35538592,0.56139533,0.78072026],
           [0.35410291,0.5539673 ,0.77033764],
           [0.35254897,0.54666505,0.75969448],
           [0.35075672,0.53945876,0.74895064],
           [0.34877758,0.53232892,0.7381625 ],
           [0.34664162,0.52526429,0.72736124],
           [0.34435551,0.51825625,0.71659672],
           [0.34194774,0.5112987 ,0.7058647 ],
           [0.33944808,0.50438565,0.69515653],
           [0.33685429,0.49751381,0.68449724],
           [0.33415774,0.49068222,0.67390715],
           [0.33139395,0.48388571,0.66335972],
           [0.32855066,0.47712407,0.65287554],
           [0.32564063,0.47039473,0.64244811],
           [0.32267383,0.46369551,0.63207233],
           [0.31964491,0.45702587,0.62175894],
           [0.31655301,0.45038485,0.61151343],
           [0.31340945,0.44377042,0.60132682],
           [0.31021415,0.43718169,0.59120234],
           [0.30696744,0.4306178 ,0.58114236],
           [0.30367607,0.42407754,0.5711394 ],
           [0.30034689,0.41755985,0.56118448],
           [0.29697042,0.41106429,0.55129497],
           [0.29354509,0.40459029,0.54147441],
           [0.290074  ,0.39813686,0.53172034],
           [0.28657704,0.39170236,0.5220016 ],
           [0.28303542,0.38528692,0.51234982],
           [0.2794477 ,0.37889001,0.50276765],
           [0.27583003,0.37251001,0.49323039],
           [0.27217781,0.3661465 ,0.48374551],
           [0.26848195,0.35979928,0.4743284 ],
           [0.26475714,0.35346682,0.46495538],
           [0.26099969,0.34714867,0.45563224],
           [0.25720016,0.34084464,0.44637494],
           [0.25337001,0.33455376,0.43716149],
           [0.2495037 ,0.32827523,0.42800385],
           [0.24559986,0.32200851,0.41890349],
           [0.24168115,0.31575112,0.40982417],
           [0.23772522,0.3095035 ,0.40080435],
           [0.23373741,0.30326418,0.39183715],
           [0.22972114,0.29703211,0.38291647],
           [0.22567889,0.29080626,0.37403743],
           [0.22159602,0.28458753,0.3652187 ],
           [0.21748806,0.27837341,0.35643725],
           [0.21334734,0.27216396,0.34770207],
           [0.20917384,0.26595834,0.33901143],
           [0.20497419,0.25975488,0.33035426],
           [0.20073609,0.25355415,0.32174591],
           [0.19646402,0.24735467,0.31317814],
           [0.19216233,0.2411549 ,0.30464273],
           [0.18782121,0.23495503,0.29615138],
           [0.18344418,0.22875353,0.28769687],
           [0.1790341 ,0.22254891,0.27927272],
           [0.17458674,0.21634052,0.27088243],
           [0.17009663,0.21012784,0.26253133],
           [0.16557175,0.20390856,0.25420519],
           [0.16100864,0.1976818 ,0.24590618],
           [0.15640125,0.19144702,0.23764019],
           [0.15175071,0.18520272,0.22940273],
           [0.14706031,0.17894709,0.2211861 ],
           [0.14232656,0.17267931,0.21299261],
           [0.13754743,0.16639847,0.2048227 ],
           [0.13271885,0.16010411,0.19668022],
           [0.12784557,0.15379483,0.18855655],
           [0.12292602,0.14747046,0.1804533 ],
           [0.11796041,0.14113116,0.17237058],
           [0.11294929,0.13477776,0.16430966],
           [0.10789316,0.12841207,0.15627396],
           [0.10279583,0.12203667,0.14826472],
           [0.09766097,0.1156557 ,0.14028679],
           [0.0924941 ,0.10927496,0.13234647],
           [0.08730275,0.1029023 ,0.12445218],
           [0.08210009,0.09654746,0.11660983],
           [0.07689606,0.09022391,0.10883739],
           [0.07170701,0.08394787,0.10115125],
           [0.06655546,0.07773869,0.09356742],
           [0.06145932,0.07162111,0.08611922],
           [0.05645253,0.06562219,0.07882605],
           [0.05156448,0.05977493,0.07172586],
           [0.04683481,0.05411596,0.06485233],
           [0.04230684,0.04868651,0.05824493],
           [0.03802186,0.04353359,0.05196125],
           [0.03422569,0.03869802,0.04603782],
           [0.03096098,0.03443234,0.04053513],
           [0.02821901,0.03082677,0.03559631],
           [0.02597978,0.02784726,0.03149059],
           [0.02423268,0.02545744,0.02814516],
           [0.02296288,0.02362385,0.02549899],
           [0.02215627,0.02231757,0.02350082],
           [0.0218106 ,0.02151429,0.0220952 ],
           [0.02191902,0.02119979,0.02125557],
           [0.02271103,0.021282  ,0.02093613],
           [0.02437281,0.02171346,0.02095901],
           [0.02668433,0.02259438,0.02138144],
           [0.02968687,0.02393883,0.02221107],
           [0.03343729,0.02576373,0.02345379],
           [0.03801144,0.02808515,0.0251087 ],
           [0.04336645,0.03092309,0.02718184],
           [0.04918858,0.03429687,0.02967838],
           [0.05543458,0.03821855,0.03259097],
           [0.06203593,0.04263494,0.03593321],
           [0.06895775,0.04729868,0.03969006],
           [0.07614815,0.05216769,0.04372403],
           [0.08355929,0.05720315,0.04789583],
           [0.09115768,0.06236649,0.05217351],
           [0.09891501,0.06762326,0.05652025],
           [0.10679701,0.072948  ,0.06091379],
           [0.11477878,0.07831728,0.06533121],
           [0.12283978,0.08371143,0.06975308],
           [0.13095958,0.08911599,0.07416717],
           [0.13912864,0.09451622,0.07855671],
           [0.1473346 ,0.09990273,0.08291342],
           [0.15556418,0.10527019,0.08723495],
           [0.16381443,0.11061139,0.09151249],
           [0.17208487,0.11592043,0.09573807],
           [0.1803702 ,0.12119568,0.09991148],
           [0.18866518,0.12643717,0.10403473],
           [0.19696938,0.13164387,0.1081066 ],
           [0.20527827,0.13681774,0.11213152],
           [0.213596  ,0.1419572 ,0.11610577],
           [0.22192631,0.14706134,0.1200267 ],
           [0.2302699 ,0.15213109,0.12389539],
           [0.23862176,0.15717055,0.12771937],
           [0.24698374,0.1621805 ,0.13149907],
           [0.25536009,0.16716055,0.13523239],
           [0.26375638,0.17210963,0.13891576],
           [0.27216413,0.17703391,0.14256093],
           [0.28058559,0.18193402,0.14616794],
           [0.2890321 ,0.18680571,0.14972671],
           [0.29749737,0.19165393,0.15324644],
           [0.30597455,0.19648399,0.15673705],
           [0.314479  ,0.20128918,0.16018358],
           [0.32300572,0.20607367,0.16359365],
           [0.33155048,0.2108412 ,0.16697405],
           [0.34012594,0.21558621,0.17031239],
           [0.34872616,0.22031324,0.1736172 ],
           [0.35734659,0.22502612,0.17689546],
           [0.36600405,0.22971666,0.18012965],
           [0.37468078,0.23439598,0.18334175],
           [0.38338407,0.23906118,0.18652505],
           [0.39212499,0.24370699,0.18966822],
           [0.40088874,0.24834297,0.19278995],
           [0.4096922 ,0.25296042,0.19587201],
           [0.41852001,0.25756921,0.19893374],
           [0.42737805,0.26216696,0.20196969],
           [0.43627708,0.26674827,0.20496847],
           [0.4452017 ,0.27132314,0.20794957],
           [0.45417181,0.27588066,0.21089066],
           [0.4631748 ,0.2804299 ,0.21379968],
           [0.47219273,0.28498394,0.21668411],
           [0.48124358,0.28953246,0.21952855],
           [0.49033354,0.29407238,0.22232852],
           [0.49944856,0.29861446,0.22508658],
           [0.50861167,0.30314405,0.2277916 ],
           [0.51779775,0.3076787 ,0.2304605 ],
           [0.52701373,0.31221492,0.23308712],
           [0.53627683,0.31674224,0.23565709],
           [0.54555894,0.32128054,0.23818929],
           [0.55487405,0.32582148,0.24067232],
           [0.56423551,0.33035722,0.24309185],
           [0.57362611,0.33490034,0.24546031],
           [0.58303858,0.33945713,0.24777857],
           [0.59248924,0.34401744,0.25003284],
           [0.60197886,0.34858209,0.25221604],
           [0.61150088,0.35315663,0.25433113],
           [0.62103657,0.35775551,0.25638662],
           [0.63059593,0.36237312,0.25837035],
           [0.6401781 ,0.36701152,0.26027646],
           [0.64977699,0.3716765 ,0.26210288],
           [0.65939731,0.37636726,0.26382643],
           [0.66902372,0.38109593,0.26545552],
           [0.67864449,0.38587263,0.26699051],
           [0.68826521,0.39069704,0.26839377],
           [0.69785415,0.39559308,0.26968908],
           [0.70737568,0.40059004,0.27087417],
           [0.71683651,0.40568677,0.27191115],
           [0.72618353,0.41092419,0.27282425],
           [0.73534543,0.41635686,0.27364734],
           [0.74425981,0.42203147,0.27442165],
           [0.75284566,0.42800539,0.27523243],
           [0.76100611,0.43434057,0.27625326],
           [0.76868037,0.44106837,0.27766548],
           [0.7758792 ,0.44816591,0.27960509],
           [0.78267704,0.45556688,0.28211764],
           [0.78916854,0.46319579,0.28517395],
           [0.79538144,0.47102871,0.28875084],
           [0.80143068,0.47898393,0.29273197],
           [0.8073008 ,0.48707099,0.29711069],
           [0.8130692 ,0.49523703,0.30180051],
           [0.81874176,0.50347998,0.30675614],
           [0.82434819,0.51177951,0.31195445],
           [0.82987976,0.52014373,0.31736339],
           [0.83536635,0.52855271,0.3229662 ],
           [0.84080257,0.53701191,0.3287411 ],
           [0.84619515,0.545518  ,0.33467258],
           [0.85155149,0.55406724,0.34074726],
           [0.85688955,0.56264906,0.34695267],
           [0.86220516,0.57126777,0.35327808],
           [0.86750495,0.57992046,0.3597141 ],
           [0.87276503,0.58862235,0.36627547],
           [0.87800636,0.59736182,0.37294251],
           [0.88324183,0.60613258,0.37970083],
           [0.88847761,0.61493228,0.38654401],
           [0.89367127,0.62378604,0.39350308],
           [0.89886443,0.6326712 ,0.40054385],
           [0.90406424,0.64158518,0.40765728],
           [0.90924383,0.65054331,0.41487146],
           [0.91442094,0.65953741,0.42216354],
           [0.91961697,0.66855709,0.42951736],
           [0.92476641,0.67763831,0.43698293],
           [0.92993584,0.68674677,0.44450693],
           [0.93510098,0.69589622,0.45211039],
           [0.94026688,0.70508474,0.45979401],
           [0.94545794,0.71430152,0.4675327 ],
           [0.95059367,0.72358869,0.47538122],
           [0.95575743,0.73290539,0.48327275],
           [0.96093411,0.74226109,0.49121058],
           [0.96611603,0.75166088,0.4991981 ],
           [0.97131142,0.76110161,0.50723372]]

cm = ListedColormap(cm_data, name="bluered_dark")
matplotlib.cm.register_cmap(name='bluered_dark', cmap=cm)
```

```
<ipython-input-8-ae72ef3db3c2>:264: MatplotlibDeprecationWarning: The register_cmap function w
as deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotli
b.colormaps.register(name)`` instead.
  matplotlib.cm.register_cmap(name='bluered_dark', cmap=cm)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-ae72ef3db3c2> in <cell line: 264>()
    262 
    263 cm = ListedColormap(cm_data, name="bluered_dark")
--> 264 matplotlib.cm.register_cmap(name='bluered_dark', cmap=cm)
    265 

/usr/local/lib/python3.10/dist-packages/matplotlib/_api/deprecation.py in wrapper(*args, **kwa
rgs)
    198         def wrapper(*args, **kwargs):
    199             emit_warning()
--> 200             return func(*args, **kwargs)
    201 
    202         old_doc = inspect.cleandoc(old_doc or '').strip('\n')

/usr/local/lib/python3.10/dist-packages/matplotlib/cm.py in register_cmap(name, cmap, override
_builtin)
    261     # the global ColormapRegistry
    262     _colormaps._allow_override_builtin = override_builtin
--> 263     _colormaps.register(cmap, name=name, force=override_builtin)
    264     _colormaps._allow_override_builtin = False
    265 

/usr/local/lib/python3.10/dist-packages/matplotlib/cm.py in register(self, cmap, name, force)
    134                 # don't allow registering an already existing cmap
    135                 # unless explicitly asked to
--> 136                 raise ValueError(
    137                     f'A colormap named "{name}" is already registered.')
    138             elif (name in self._builtin_cmaps

ValueError: A colormap named "bluered_dark" is already registered.
```

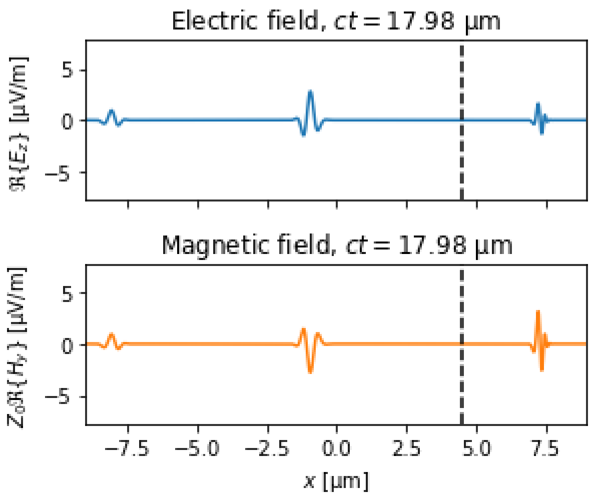
In [ ]:

```python
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.colors import ListedColormap
import matplotlib.cm
#import bluered_dark
import time

plt.rcParams.update({
    'figure.figsize': (12 / 2.54, 9 / 2.54),
    'figure.subplot.bottom': 0.15,
    'figure.subplot.left': 0.165,
    'figure.subplot.right': 0.90,
    'figure.subplot.top': 0.9,
    'axes.grid': False,
    'image.cmap': 'bluered_dark',
})
```

```python
# save figures to disk
save_figures = False
# save movie to disk (requires Ffmpeg)
save_movie = False

plt.close('all')

# constants
c = 2.99792458e8  # speed of light [m/s]
mu0 = 4 * np.pi * 1e-7  # vacuum permeability [Vs/(Am)]
epsilon_0 = 1 / (mu0 * c ** 2)  # vacuum permittivity [As/(Vm)]
Z0 = np.sqrt(mu0 / epsilon_0)  # vacuum impedance [Ohm]

# geometry parameters
x_width = 18e-6  # x_width of computatinal domain [m]
n1 = 1  # refractive index in front of interface
n2 = 2  # refractive index behind interface
interf_pos = x_width / 4  # postion of dielectric interface

# simulation parameters
dx = 15e-9 # grid spacing [m]
dt = 60e-15  # duration of simulation [s]

Nx = int(round(x_width / dx)) + 1  # number of grid points

# source parameters
source_frequency = 500e12  # [Hz]
source_position = 0  # [m]
source_pulse_length = 1e-15  # [s]

def fdtd_1d(eps_rel, dx, dt, source_frequency, source_position,
            source_pulse_length):
    """Computes the temporal evolution of a pulsed excitation using the
    1D FDTD method. The temporal center of the pulse is placed at a
    simulation time of 3*source_pulse_length. The origin x=0 is in the
    center of the computational domain. All quantities have to be
    specified in SI units.

    Arguments
    ---------
        eps_rel : 1d-array
            Rel. permittivity distribution within the computational domain.
        dx : float
            Spacing of the simulation grid (please ensure dx <= lambda/20).
        time_span : float
            Time span of simulation.
        source_frequency : float
            Frequency of current source.
        source_position : float
            Spatial position of current source.
        source_pulse_length :
            Temporal width of Gaussian envelope of the source.

    Returns
    -------
        Ez : 2d-array
            Z-component of E(x,t) (each row corresponds to one time step)
        Hy : 2d-array
            Y-component of H(x,t) (each row corresponds to one time step)
        x  : 1d-array
            Spatial coordinates of the field output
        t  : 1d-array
            Time of the field output
    """


    # calculate time step and prefactors
    del_t = dx / (2 * c)
    e_factor =del_t / epsilon_0
    h_factor =del_t / mu0

    # create position and time vectors
    Nx = eps_rel.size
    x = np.arange(Nx) * dx - (Nx - 1) / 2.0 * dx
    Numb_iter = int(round(dt /del_t))
    t = np.arange(Numb_iter + 1) * del_t

    # allocate field arrays
    Ez = np.zeros((Numb_iter + 1, Nx), dtype=complex)
    Hy = np.zeros((Numb_iter + 1, Nx - 1), dtype=complex)

    # source properties
    # angular frequency (avoids multiplication by 2*pi every iteration)
    source_angular_frequency = 2 * np.pi * source_frequency
    # time offset of pulse center
    t0 = 3 * source_pulse_length
    # x-grid index of delta-source (rounded to nearest grid point)
    source_ind = int(round((source_position - x[0]) / dx))
    if (source_ind < 1) or (source_ind > Nx - 2):
        raise ValueError('Source position out of range')

    for n in range(0, Numb_iter):
        # calculate E at time n + 1, the values at the spatial indices
        # 0 and Nx -1 are determined by the PEC boundary conditions and don't need to be upda
        Ez[n + 1, 1:-1] = (Ez[n, 1:-1]
                + e_factor / dx * (Hy[n, 1:] - Hy[n, :-1]) / eps_rel[1:-1])

        # add source term to Ez
        # source current has to  be taken at n + 1/2
        t_source = (n + 0.5) * del_t - t0
        j_source = (np.exp(-1j * source_angular_frequency * t_source)  # carrier
                  * np.exp(-(t_source / source_pulse_length) ** 2))  # envelope
        Ez[n + 1, source_ind] -= e_factor / eps_rel[source_ind] * j_source

        # calculate H at time n + 3/2
        Hy[n + 1, :] = Hy[n, :] + h_factor / dx * (Ez[n + 1, 1:] - Ez[n + 1, :-1])

    # The fields are returned on the same x-grid as eps_rel whereby
    # both Ez and Hy are returned at the same points in space and time
    # (the user should not need to care about the peculiarities of
    # the Yee grid and the leap frog algorithm).

    # interpolate Hy to same t and x grid as Ez
    Hy[1:, :] = 0.5 * (Hy[:-1, :] + Hy[1:, :])
    Hy = avg_axes(repl_bound_val(Hy, [1]), [1])
    return Ez, Hy, x, t

def avg_axes(field, axes):
    """Averages neighboring values of the given field along the specified axes

    Arguments
    ---------
        field : nd-array
            Field array to be averaged.
        axes : sequence
            Sequence of axes that shall be averaged.
    Returns
    -------
        res: nd-array
            Field averaged along the specified axes.
    """

    # prefactor for calculating avergaes with the same numerical preciosion
    # as the input field.
    f = np.array(0.5, dtype=field.dtype)

    c = slice(0, None)  # full
    l = slice(0, -1)  # left part of average
    r = slice(1, None)  # right part of average
    res = field
    for ax in axes:
        a = tuple((l if i == ax else c for i in range(res.ndim)))
        b = tuple((r if i == ax else c for i in range(res.ndim)))
        res = f * (res[a] + res[b])
    return res




def repl_bound_val(field, axes):
    """Replicates the boundary values of the given field along the
    specified axes

    Arguments
    ---------
        field : nd-array
            Field array to be padded.
        axes : sequence
            Sequence of axes that shall be padded.

    Returns
    -------
        res: nd-array
            Field padded along the specified axes.
    """
    res = field
    for ax in axes:
        a = tuple((0 if i == ax else slice(None)
                   for i in range(field.ndim)))
        b = tuple((-1 if i == ax else slice(None)
                   for i in range(field.ndim)))
        na = tuple((np.newaxis if i == ax else slice(None)
                    for i in range(field.ndim)))
        res = np.concatenate((res[a][na], res, res[b][na]), axis=ax)
    return res

class Timer(object):
    """Tic-toc timer.
    """

    def __init__(self):
        """Initializer.
        Stores the current time.
        """
        self._tic = time.time()

    def tic(self):
        """Stores the current time.
        """
        self._tic = time.time()

    def toc(self):
        """Returns the time in seconds that has elapsed since the last call
        to tic().
        """
        return time.time() - self._tic
```

Animation for temporal evolution of electric and magnetic fields: (the code generating animations has been commented out. A Snapshot in included below the code)

In [ ]:
```
# Animation class
import matplotlib.animation as animation
class Fdtd1DAnimation(animation.TimedAnimation):
    def __init__(self, x, t, Ez, Hy, interf_pos=None):
        self.c = 2.99792458e8
        self.Z0 = np.sqrt(mu0 / epsilon_0)
        self.Ez = Ez
        self.Hy = Hy
        self.x = x * 1e6
        self.ct = self.c * t * 1e6
        self.frame_step = int(round((t[1] - t[0]) / (2e-15 / 25)))
        if self.frame_step == 0:
            self.frame_step = 1
        colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
        vmax = max(np.max(np.abs(Ez)), np.max(np.abs(Hy)) * self.Z0) * 1e6
        fig, ax = plt.subplots(2, 1, sharex=True, gridspec_kw={'hspace': 0.4})
        self.line_E, = ax[0].plot(self.x, self.E_at_step(0), color=colors[0], label='$\\Re\\{
        self.line_H, = ax[1].plot(self.x, self.H_at_step(0), color=colors[1], label='$Z_0\\Re
        if interf_pos is not None:
            for a in ax:
                a.axvline(interf_pos * 1e6, ls='--', color='k')
        for a in ax:
            a.set_xlim(self.x[0], self.x[-1])
            a.set_ylim(-1.1 * vmax, 1.1 * vmax)
        ax[0].set_ylabel('$\\Re\\{E_z\\}$ [µV/m]')
        ax[1].set_ylabel('$Z_0\\Re\\{H_y\\}$ [µV/m]')
        self.text_E = ax[0].set_title('')
        self.text_H = ax[1].set_title('')
        ax[1].set_xlabel('$x$ [µm]')
        super().__init__(fig, interval=1000 / 25, blit=False)

    def E_at_step(self, n):
        return self.Ez[n, :].real * 1e6

    def H_at_step(self, n):
        return self.Z0 * self.Hy[n, :].real * 1e6

    def new_frame_seq(self):
        return iter(range(0, self.ct.size, self.frame_step))

    def _init_draw(self):
        self.line_E.set_ydata(self.x * np.nan)
        self.line_H.set_ydata(self.x * np.nan)
        self.text_E.set_text('')
        self.text_H.set_text('')

    def _draw_frame(self, framedata):
        i = framedata
        self.line_E.set_ydata(self.E_at_step(i))
        self.line_H.set_ydata(self.H_at_step(i))
        self.text_E.set_text('Electric field, $ct = {0:1.2f}$ µm'.format(self.ct[i]))
        self.text_H.set_text('Magnetic field, $ct = {0:1.2f}$ µm'.format(self.ct[i]))
        self._drawn_artists = [self.line_E, self.line_H, self.text_E, self.text_H]

# Make video
#fps = 25
#step = t[-1] / fps / 30
#anim = Fdtd1DAnimation(x, t, Ez, Hy, interf_pos)
#Writer = animation.PillowWriter
#writer = Writer(fps=fps)
#anim.save('fdtd_1d.gif', writer=writer)

#plt.show()
```



This image is a snapshot of the temporal evolution of electric and magnetic field from the animations generated by the preceeding code block. Located at x = 0 µm is the pulsed current source, and at x = 4.5 µm is the interface represented by a dashed black line.

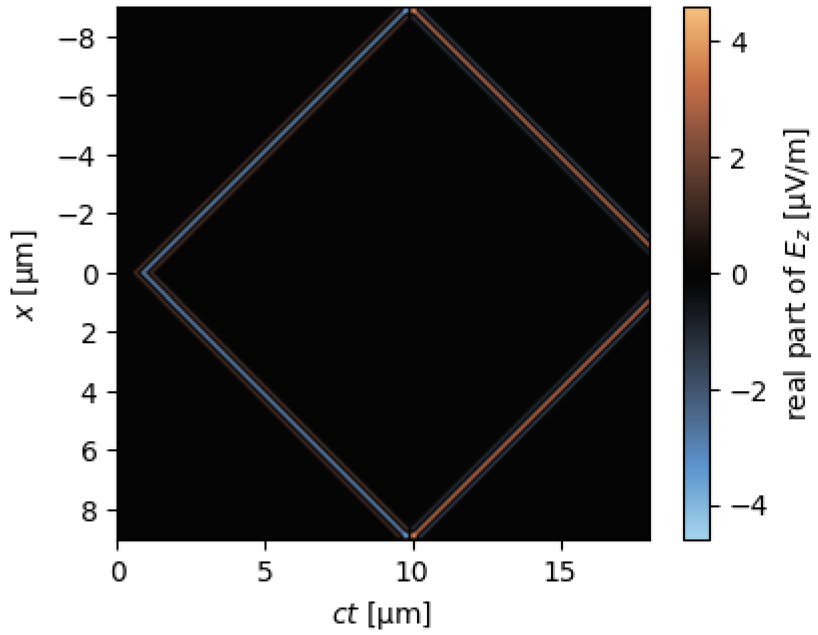
Task 2.1: Propagation of pulse through homogeneous and inhomogeneous medium in 1D

```
In [ ]: # timer to measure the execution time
timer = Timer()
# simulate homogeneous medium
eps_rel = np.ones((Nx,)) * n1 ** 2
Ez_ref, Hy_ref, x, t = fdtd_1d(eps_rel, dx, dt,
                               source_frequency, source_position,
                               source_pulse_length)

# simulate dielectric interface
eps_rel[x >= interf_pos] = n2 ** 2
timer.tic()
Ez, Hy, x, _ = fdtd_1d(eps_rel, dx, dt,
                       source_frequency, source_position,
                       source_pulse_length)
print('time: {0:g}s'.format(timer.toc()))

#  plot time traces of electric field and Poynting vector
extent = [t[0] * c * 1e6, t[-1] * c * 1e6, x[-1] * 1e6, x[0] * 1e6]

# plot electric field
vmax = max(np.max(np.abs(Ez_ref)), np.max(np.abs(Ez))) * 1e6
fig = plt.figure()
```

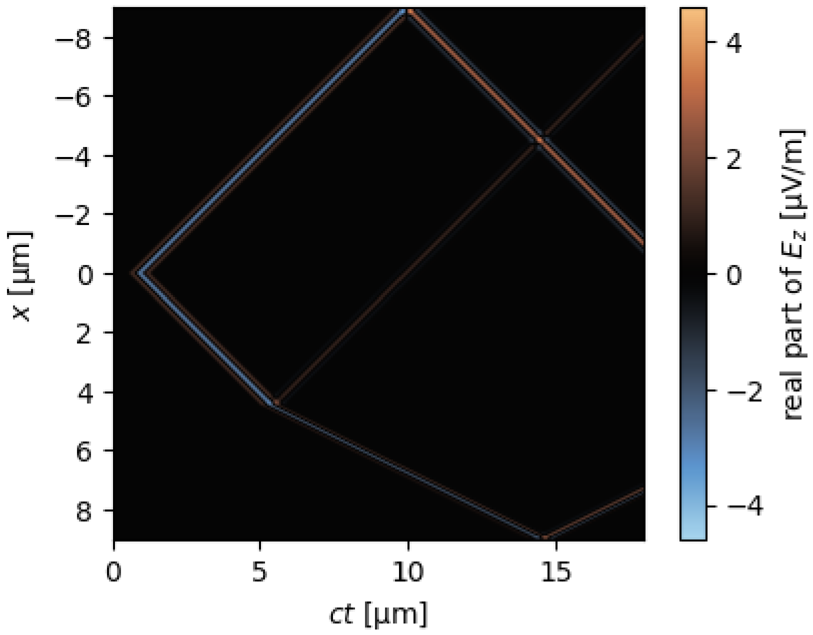
```python
plt.imshow(Ez_ref.real.T * 1e6, extent=extent, vmin=-vmax, vmax=vmax)
cb = plt.colorbar()
plt.xlabel('$ct$ [µm]')
plt.ylabel('$x$ [µm]')
plt.title('Propagation through 1D homogeneous space' )
cb.set_label('real part of $E_z$ [µV/m]')
if save_figures:
    fig.savefig('Ez_ref_1D.pdf', dpi=300)

fig = plt.figure()
plt.imshow(Ez.real.T * 1e6, extent=extent, vmin=-vmax, vmax=vmax)
cb = plt.colorbar()
plt.xlabel('$ct$ [µm]')
plt.ylabel('$x$ [µm]')
plt.title('Propagation through 1D inhomogeneous space')
cb.set_label('real part of $E_z$ [µV/m]')
if save_figures:
    fig.savefig('Ez_1D.pdf', dpi=300)

# calculate Poynting vector
Sx_ref = -0.5 * np.real(Ez_ref * np.conj(Hy_ref))
Sx = -0.5 * np.real(Ez * np.conj(Hy))

# plot power flow
vmax = np.abs(Sx_ref).max()
fig = plt.figure()
plt.imshow(Sx_ref.T / vmax, extent=extent, vmin=-1, vmax=1)
cb = plt.colorbar()
plt.xlabel('$ct$ [µm]')
plt.ylabel('$x$ [µm]')
cb.set_label('normalized Poynting vector')
plt.title('For 1D homogeneous space')
cb.set_ticks(np.linspace(-1, 1, 11))
if save_figures:
    fig.savefig('Sx_ref_1D.pdf', dpi=300)

fig = plt.figure()
plt.imshow(Sx.T / vmax, extent=extent, vmin=-1, vmax=1)
cb = plt.colorbar()
plt.xlabel('$ct$ [µm]')
plt.ylabel('$x$ [µm]')
cb.set_label('normalized Poynting vector')
plt.title('For 1D inhomogeneous space')
cb.set_ticks(np.linspace(-1, 1, 11))
plt.show()

if save_figures:
    fig.savefig('Sx_1D.pdf', dpi=300)
```

```
time: 0.226006s
```

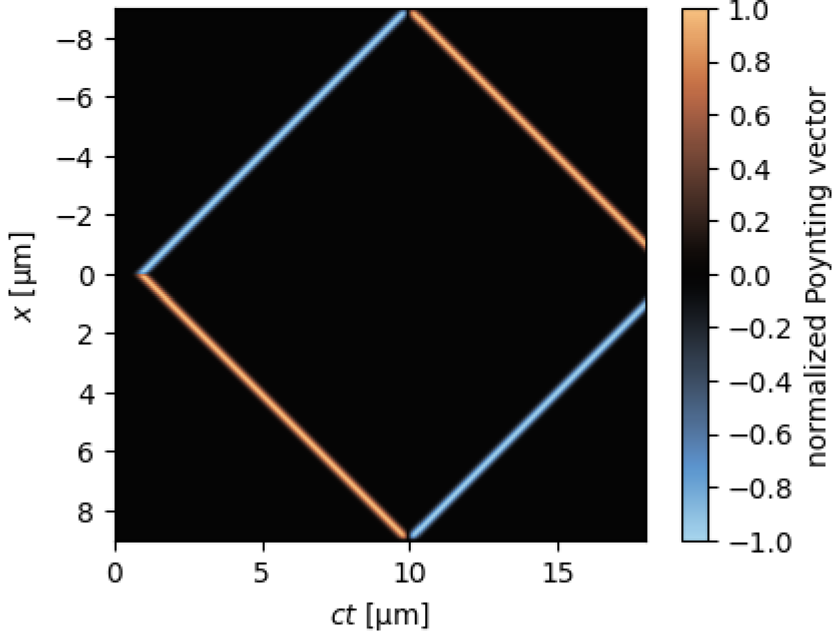
Propagation through 1D homogeneous space

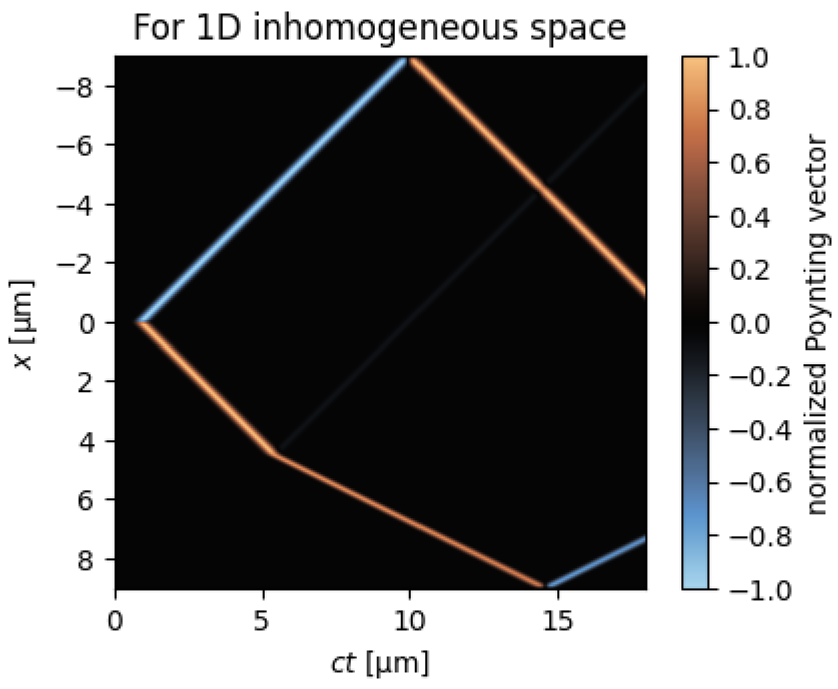


Propagation through 1D inhomogeneous space



For 1D homogeneous space





Analysis:

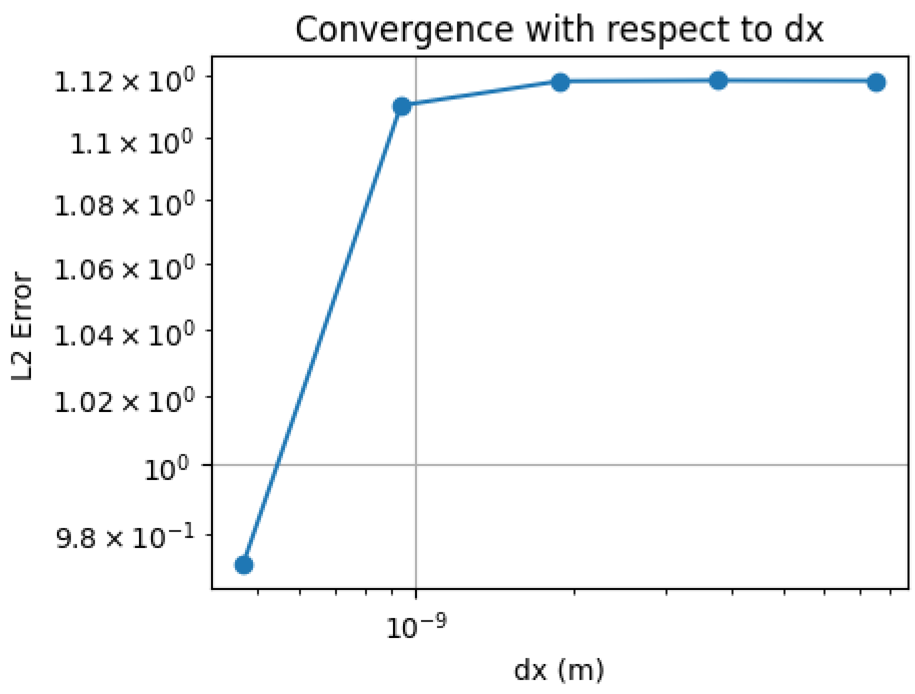
(i) While pulse is being propagated through a homogeneous space, the first and 3rd plots show the time trace of the real part of Electric field E_z and the x-component of normalized Poynting vector S_x respectively in one spatial and one temporal domain. Both a forward and backward travelling wave are excited because the pulsed current source does not have a preference in direction of emission. A symmetric pulse profile is generated. Propagation of the pulse does not change its shape since so dispersion occurs in the case of a homogeneous space. The pulses are perfectly reflected from the computational domain boundaries due to the perfectly electric conducting walls condition. As a result, the tangential electric field suffers a change in sign, as does the Poynting vector.

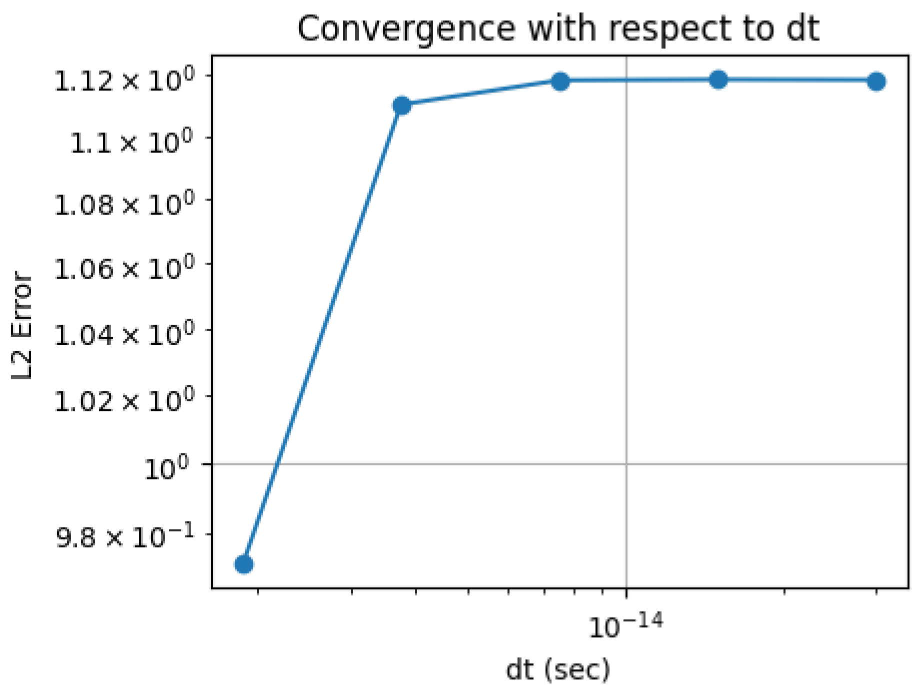
(ii) In the case of a pulse propagating through an inhomogeneous spce with an iterface as described above, the 3rd and 4th plots show time trace foe the real part the electric field E_z, and the x-component of the normalized Poynting vector S_x. The forward travelling pulse suffers partial reflectio at the interface. As compared to slope of the time traces in n_1, slope of the time traces in n_2 are flatter on account of the reduced group velocity.

Task 3.1: Testing convergence and accuracy of obtained results (from 1D implementaion) vs. parameters dx and dt

To test the convergence of the results with respect to the grid spacing dx and the time step dt, we systematically reduce these parameters and examine how the computed fields change. We calculate some norm of the difference between the results obtained with different dx anddt settings. This can provide a quantitative measure of how the solution converges as the discretization is refined.The codes are shwown bwlow:

In [ ]:
```
from scipy.interpolate import interp1d
#import fdtd_1d

def convergence_test(fdtd_1d_func, initial_dx, initial_dt, n1, n2, source_frequency, source_p
    # Store initial dx and dt
    dx = initial_dx
    dt = initial_dt

    # Initialize list to hold errors
    dx_list = []
    dt_list = []
    errors = []

    # Initial simulation with initial parameters
    Nx = int(round(x_width / dx)) + 1
    eps_rel = np.ones((Nx,)) * n1 ** 2
    eps_rel[int(round((interf_pos / dx))):] = n2 ** 2
    _, Hy_prev, x_prev, _ = fdtd_1d_func(eps_rel, dx, dt, source_frequency, source_position,

    for _ in range(num_tests):
        # Update dx and dt
        dx *= factor
        dt *= factor
        Nx = int(round(x_width / dx)) + 1
        eps_rel = np.ones((Nx,)) * n1 ** 2
        eps_rel[int(round((interf_pos / dx))):] = n2 ** 2

        # Run simulation
        _, Hy, x, _ = fdtd_1d_func(eps_rel, dx, dt, source_frequency, source_position, source_

        # Interpolate Hy_prev to the new x grid
        interpolate = interp1d(x_prev, Hy_prev[-1], kind='cubic', fill_value="extrapolate")
        Hy_prev_interpolated = interpolate(x)

        # Calculate error (L2 norm of the difference)
        error = np.linalg.norm(Hy[-1] - Hy_prev_interpolated) / np.linalg.norm(Hy_prev_interp

        # Store results
        dx_list.append(dx)
        dt_list.append(dt)
        errors.append(error)

        # Update previous solution
        Hy_prev = Hy
        x_prev = x

    return dx_list, dt_list, errors


# Define initial parameters for the convergence test
initial_dx = 15e-9  # initial grid spacing [m]
initial_dt = 60e-15  # initial duration of simulation [s]

# Run the convergence test
dx_list, dt_list, errors = convergence_test(fdtd_1d, initial_dx, initial_dt, n1, n2, source_f

# Plot results
plt.figure()
plt.loglog(dx_list, errors, 'o-')
plt.xlabel('dx (m)')
plt.ylabel('L2 Error')
plt.title('Convergence with respect to dx')
plt.grid(True)
#plt.show()

plt.figure()
plt.loglog(dt_list, errors, 'o-')
plt.xlabel('dt (sec)')
plt.ylabel('L2 Error')
plt.title('Convergence with respect to dt')
plt.grid(True)
#plt.show()
```





This graph shows an initial sharp decrease in error as dt decreases, which then plateaus. This behavior is typical in numerical simulations where reducing the time step initially leads to significant improvements in accuracy. Once the time step becomes sufficiently small, other factors such as numerical precision, spatial discretization errors (determined by dx), or inherent stability limits of the numerical method may dominate, leading to a plateau in the convergence curve. This indicates that further reduction in dt without a corresponding adjustment in dx or improvements in the numerical scheme will not yield significantly better accuracy.

Similarly, the convergence with respect to dx shows a sharp decrease in error as dx is reduced, followed by a plateau. This suggests that the spatial discretization is initially coarse and reducing dx significantly improves the solution's accuracy.

There are some possible reasons: firstly, dt is not sufficiently small compared to dx, the temporal discretization error may limit further improvements. Secondly, at very small dx, numerical stability issues or floating-point precision limits may arise. Finally, The physical model or boundary conditions used might have inherent limitations that prevent further improvements in accuracy beyond a certain point.

Task 1.2 Implementing the FDTD method in 3D:

In 3D case, we have three equations derived from Amper's law:

$$\frac{\partial H_z(r,t)}{\partial y} - \frac{\partial H_y(r,t)}{\partial z} = \epsilon_0 \epsilon_r(r) \frac{\partial E_x(r,t)}{\partial t} + j_x(r,t)$$

$$\frac{\partial H_x(r,t)}{\partial z} - \frac{\partial H_z(r,t)}{\partial x} = \epsilon_0 \epsilon_r(r) \frac{\partial E_y(r,t)}{\partial t} + j_y(r,t)$$

$$\frac{\partial H_y(r,t)}{\partial x} - \frac{\partial H_x(r,t)}{\partial y} = \epsilon_0 \epsilon_r(r) \frac{\partial E_z(r,t)}{\partial t} + j_z(r,t)$$

And derived from Farady's law, we have:

$$\frac{\partial E_z(r,t)}{\partial y} - \frac{\partial E_y(r,t)}{\partial z} = -\mu_0 \frac{\partial H_x(r,t)}{\partial t}$$

$$\frac{\partial E_x(r,t)}{\partial z} - \frac{\partial E_z(r,t)}{\partial x} = -\mu_0 \frac{\partial H_y(r,t)}{\partial t}$$

$$\frac{\partial E_y(r,t)}{\partial x} - \frac{\partial E_x(r,t)}{\partial y} = -\mu_0 \frac{\partial H_z(r,t)}{\partial t}$$

Similar to the 1D case, these equations can be discretized on a Yee grid, and by approximating the derivatives with finite differences, the electric field components can be expressed as follows:

$$E_x|_{i+0.5,j,k}^{n+1} = E_x|_{i+0.5,j,k}^{n} - \frac{\Delta t}{\epsilon_0 \epsilon_{i+0.5,j,k}} [\frac{H_z|_{i+0.5,j+0.5,k}^{n+0.5} - H_z|_{i+0.5,j-0.5,k}^{n+0.5}}{\Delta y} - \frac{H_y|_{i+0.5,j,k+0.5}^{n+0.5} - H_y|_{i+0.5,j,k-0.5}^{n+0.5}}{\Delta z} - j_x|_{i+0.5,j,k}^{n+0.5}$$

$$E_y|_{i,j+0.5,k}^{n+1} = E_y|_{i,j+0.5,k}^{n} - \frac{\Delta t}{\epsilon_0 \epsilon_{i,j+0.5,k}} [\frac{H_x|_{i,j+0.5,k+0.5}^{n+0.5} - H_x|_{i,j+0.5,k-0.5}^{n+0.5}}{\Delta z} - \frac{H_z|_{i+0.5,j+0.5,k}^{n+0.5} - H_z|_{i-0.5,j+0.5,k}^{n+0.5}}{\Delta x} - j_y|_{i,j+0.5,k}^{n+0.5}$$

$$E_z|_{i,j,k+0.5}^{n+1} = E_x|_{i,j,k+0.5}^{n} - \frac{\Delta t}{\epsilon_0 \epsilon_{i,j,k+0.5}} [\frac{H_y|_{i+0.5,j,k+0.5}^{n+0.5} - H_y|_{i-0.5,j,k+0.5}^{n+0.5}}{\Delta x} - \frac{H_x|_{i,j+0.5,k+0.5}^{n+0.5} - H_x|_{i,j-0.5,k+0.5}^{n+0.5}}{\Delta y} - j_z|_{i,j,k+0.5}^{n+0.5}$$

where

$$\frac{1}{\epsilon_{i+0.5,j,k}} = \frac{1}{2}(\frac{1}{\epsilon_{i,j,k}} + \frac{1}{\epsilon_{i+1,j,k}})$$

$$\frac{1}{\epsilon_{i,j+0.5,k}} = \frac{1}{2}(\frac{1}{\epsilon_{i,j,k}} + \frac{1}{\epsilon_{i,j+1,k}})$$

$$\frac{1}{\epsilon_{i,j,k=0.5}} = \frac{1}{2}(\frac{1}{\epsilon_{i,j,k}} + \frac{1}{\epsilon_{i,j,k+1}})$$

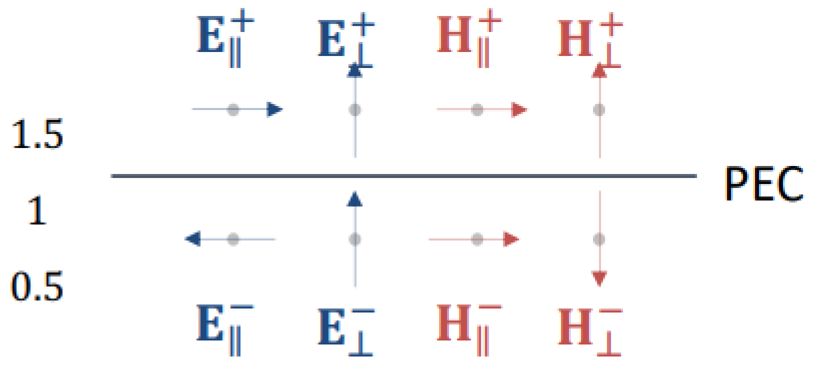
Similarly, the magnetic field components can be expressed as follows:

$$H_x|_{i,j+0.5,k+0.5}^{n+1.5} = H_x|_{i,j+0.5,k+0.5}^{n+0.5} - \frac{\Delta t}{\mu_0} [\frac{E_z|_{i,j+1,k+0.5}^{n+1} - E_z|_{i,j,k+0.5}^{n+1}}{\Delta y} - \frac{E_y|_{i,j+0.5,k+1}^{n+1} - E_y|_{i,j+0.5,k}^{n+1}}{\Delta z}]$$

$$H_y|_{i+0.5,j,k+0.5}^{n+1.5} = H_y|_{i+0.5,j,k+0.5}^{n+0.5} - \frac{\Delta t}{\mu_0} [\frac{E_x|_{i+0.5,j,k+1}^{n+1} - E_x|_{i+0.5,j,k}^{n+1}}{\Delta z} - \frac{E_z|_{i+1,j,k+0.5}^{n+1} - E_z|_{i,j,k+0.5}^{n+1}}{\Delta x}]$$

$$H_z|_{i+0.5,j+0.5,k}^{n+1.5} = H_z|_{i+0.5,j+0.5,k}^{n+0.5} - \frac{\Delta t}{\mu_0} [\frac{E_y|_{i+1,j+0.5,k}^{n+1} - E_y|_{i,j+0.5,k}^{n+1}}{\Delta x} - \frac{E_x|_{i+0.5,j+1,k}^{n+1} - E_x|_{i+0.5,j,k}^{n+1}}{\Delta y}]$$

PEC boundary condition



In Finite-Difference Time-Domain (FDTD) simulations, handling boundary conditions accurately is crucial for maintaining the physical correctness of the simulation. When dealing with Perfect Electric Conductor (PEC) boundaries, specific mirror symmetries are utilized to interpolate the missing values of the electric and magnetic fields at the boundaries. Here's an explanation of the physical reasons for these mirror symmetries:

PEC Boundary Conditions A Perfect Electric Conductor (PEC) is an idealized material that reflects all incident electromagnetic waves without any loss. At the surface of a PEC, the following conditions are always true:

1.The tangential component of the electric field must be zero This implies that the electric field vector parallel to the surface of the PEC is always zero.

2.The normal component of the electric field can exist, but it must satisfy boundary conditions that enforce continuity and reflection properties The normal component just outside the PEC is equal and opposite to the normal component just inside

3.The tangential component of the magnetic field must be continuous across the boundary:

4.The normal component of the magnetic field must be zero:This implies that the magnetic field vector perpendicular to the surface of the PEC is always zero.

Therefore,we could get the relation below:

Electric Field (E):

Parallel Components: $E \parallel - = - E \parallel +$

Normal Components: $E \perp - = E \perp +$

Magnetic Field (H):

Parallel Components: $H \parallel - = H \parallel +$

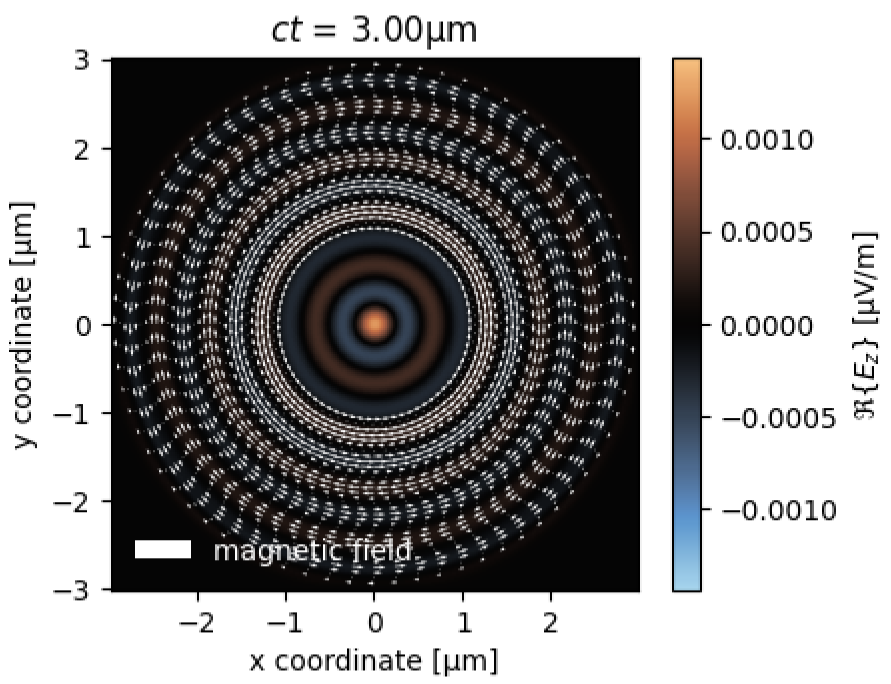
Normal Components: $H \perp - = - H \perp +$

3D FDTD functoin and implementation

In [ ]:
```python
import numpy as np
from matplotlib import pyplot as plt
import time
from scipy.interpolate import interpn
def average_axes(field, axes):


    # prefactor for calculating avergaes with the same numerical preciosion
    # as the input field.
    f = np.array(0.5, dtype=field.dtype)

    c = slice(0, None)  # full
    l = slice(0, -1)  # left part of average
    r = slice(1, None)  # right part of average
    res = field
    for ax in axes:
        a = tuple((l if i == ax else c for i in range(res.ndim)))
        b = tuple((r if i == ax else c for i in range(res.ndim)))
        res = f * (res[a] + res[b])
    return res
def fdtd_3d_interpolate_field(field, z_ind, component):
    component = component.lower()
    if component == 'ex':
        rep_axes = [0]
        pad_axes = [1, 2]
    elif component == 'ey':
        rep_axes = [1]
        pad_axes = [0, 2]
    elif component == 'ez':
        rep_axes = [2]
        pad_axes = [0, 1]
    elif component == 'hx':
        rep_axes = [1, 2]
        pad_axes = [0]
    elif component == 'hy':
        rep_axes = [0, 2]
        pad_axes = [1]
    elif component == 'hz':
        rep_axes = [0, 1]
        pad_axes = [2]
    else:
        raise ValueError('Invalid field component')
    res = average_axes(
        replicate_boundary_values(
            boundary_values(field, pad_axes), rep_axes),
        rep_axes)
    return res[:, :, z_ind]


def replicate_boundary_values(field, axes):

    res = field
    for ax in axes:
        a = tuple((0 if i == ax else slice(None)
                   for i in range(field.ndim)))
        b = tuple((-1 if i == ax else slice(None)
                   for i in range(field.ndim)))
        na = tuple((np.newaxis if i == ax else slice(None)
                    for i in range(field.ndim)))
        res = np.concatenate((res[a][na], res, res[b][na]), axis=ax)
    return res


def boundary_values(field, axes):

    res = field
    for ax in axes:
        shape = tuple((1 if i == ax else n for i, n in enumerate(res.shape)))
        pad = np.zeros(shape, dtype=res.dtype)
        res = np.concatenate((pad, res, pad), axis=ax)
    return res

def fdtd_3d(eps_rel, dr, time_span, freq, tau, jx, jy, jz,field_component, z_ind, output_step
    c = 2.99792458e8
    mu0 = 4 * np.pi * 1e-7
    eps0 = 1 / (mu0 * c ** 2)
    #the eps_rel should be the grid size for 3d fdtd implementation
    Nx, Ny, Nz = eps_rel.shape
    #preset the factor used in defintion
    #time space  dr is the gird spacing here
    dt=np.float32(dr/(2*c))
    dr=np.float32(dr)
    #factor eps and mu
    E=np.float32(dt/eps0)
    M=np.float32(dt/mu0)
    #get the iteration number for the for the time
    Niter = int(round(time_span /dt/ output_step) * output_step)
    #arrange the actual time for each iteration time
    t = np.arange(0, Niter + 1, output_step) * dt
        #use the inversion of permittivity in fdtd method
    eps_rel=np.float32(1)/eps_rel
    #interpolating method
    #we need to use the interpolated as mention in the seminar in each direction
    iepsx = average_axes(eps_rel, [0])
    iepsy = average_axes(eps_rel, [1])
    iepsz = average_axes(eps_rel, [2])
    #after getting the interpolated value,deleting the original eps_rel
    del eps_rel
    #use the same method to get the interpolated value of sources
    jx = average_axes(jx, [0])
    jy = average_axes(jy, [1])
    jz = average_axes(jz, [2])

    # preset the E component in each direction according to the array sizes in the seminar
    Ex = np.zeros((Nx - 1, Ny, Nz), dtype=np.complex64)
    Ey = np.zeros((Nx, Ny - 1, Nz), dtype=np.complex64)
    Ez = np.zeros((Nx, Ny, Nz - 1), dtype=np.complex64)
    # preset the E component in each direction according to the array sizes in the seminar
    Hx = np.zeros((Nx, Ny - 1, Nz - 1), dtype=np.complex64)
    Hy = np.zeros((Nx - 1, Ny, Nz - 1), dtype=np.complex64)
    Hz = np.zeros((Nx - 1, Ny - 1, Nz), dtype=np.complex64)
    #get the slice from 1 to N-2
    Ax = slice(1, Nx - 1)
    Ay = slice(1, Ny - 1)
    Az = slice(1, Nz - 1)
    #get the slice from 0 to N-3
    Bx = slice(0, Nx - 2)
    By = slice(0, Ny - 2)
    Bz = slice(0, Nz - 2)
    #get the slice from 0 to N-2
    Cx = slice(0, Nx - 1)
    Cy = slice(0, Ny - 1)
    Cz = slice(0, Nz - 1)
    #get the slice from 1 to N-1
    Dx = slice(1, Nx)
    Dy = slice(1, Ny)
    Dz = slice(1, Nz)
    #get the slice for specific time spep in the x and y direction
    St=np.zeros((t.size,Nx,Ny),dtype=np.complex64)
    #prefactor for calculation of averages
    f = np.float32(0.5)
    F = np.zeros((t.size, Nx, Ny), dtype=np.complex64)
    next_out = 1
    #next slice index in output St
    Nextis=1
 #for time iteration ,use the for loop to do the iteration
    for n in range(Niter):
#firstly,the source need to be define
        #the time dependent part of the source,3 is initial time position
        jt=(n+0.5)*dt-3*tau
        #source term according to the formula in the seminar
        js=np.complex64(E*np.exp(-2j*np.pi*freq*jt)*np.exp(-((n+0.5)*dt-3*tau/tau)**2))
#update the Ex field here
        U =E/ dr * (Hz[Cx, Ay,Az] - Hz[Cx,By,Az])
        U -=E/ dr * (Hy[Cx, Ay,Az] - Hy[Cx, Ay, Bz])
        U -= js * jx[Cx, Ay,Az]
         # divide by eps_rel (interpolated to Ex grid)
        U *= iepsx[Cx, Ay,Az]
         # + Ex(t=n*dt)
        U += Ex[Cx, Ay,Az]
        if ((n + 1) % output_step == 0) and (field_component == 'ex'):
            F[next_out, :, :] = fdtd_3d_interpolate_field(U, z_ind,
                                                          field_component)
            next_out += 1
        Ex[Cx,Ay,Az] = U
 #update the Ey field here
        U =E/ dr * (Hx[Ax, Cy,Az] - Hx[Ax,Cy,Bz])
        U -=E/ dr * (Hz[Ax,Cy,Az] - Hz[Bx,Cy,Az])
        U -= js * jy[Ax,Cy,Az]
        U *= iepsy[Ax,Cy,Az]
        U += Ey[Ax, Cy,Az]
        if ((n + 1) % output_step == 0) and (field_component == 'ey'):
             F[next_out, :, :] = fdtd_3d_interpolate_field(U, z_ind,
                                                            field_component)
             next_out += 1
        Ey[Ax,Cy,Az] = U
#update the Ez field here
        U =E/ dr * (Hy[Ax, Ay, Cz] - Hy[Bx, Ay,Cz])
        U -=E/ dr * (Hx[Ax, Ay,Cz] - Hx[Ax, By,Cz])
        U -=js * jz[Ax,Ay,Cz]
        U *= iepsz[Ax, Ay,Cz]
        U += Ez[Ax, Ay,Cz]
        if ((n + 1) % output_step == 0) and (field_component == 'ez'):
             F[next_out, :, :] = fdtd_3d_interpolate_field(U, z_ind,
                                                            field_component)
             next_out += 1
        Ez[Ax, Ay,Cz] = U
#update the Hx field here
        U = Hx[Ax, Cy,Cz]
        U -=M/ dr * (Ez[Ax, Dy,Cz] - Ez[Ax,Cy,Cz])
        U += M / dr * (Ey[Ax,Cy, Dz] - Ey[Ax,Cy,Cz])
        if ((n + 1) % output_step == 0) and (field_component == 'hx'):
             F[next_out, :, :] = fdtd_3d_interpolate_field(
                 f * (U + Hx[Ax, Cy,Cz]), z_ind,
                 field_component)
             next_out += 1
        Hx[Ax,Cy,Cz] = U
#update the Hy field here
        U = Hy[Cx,Ay,Cz]
        U -= M/ dr * (Ex[Cx, Ay,Dz] - Ex[Cx,Ay,Cz])
        U += M/ dr * (Ez[Dx,Ay,Cz] - Ez[Cx, Ay,Cz])
        if ((n + 1) % output_step == 0) and (field_component == 'hy'):
             F[next_out, :, :] = fdtd_3d_interpolate_field(
                 f * (U + Hy[Cx,Ay,Cz]), z_ind,
                 field_component)
             next_out += 1
        Hy[Cx,Ay,Cz] = U
#update the Hz field here
        U = Hz[Cx,Cy,Az]
        U -=M / dr * (Ey[Dx,Cy,Az] - Ey[Cx, Cy,Az])
        U +=M/ dr * (Ex[Cy,Dy,Az] - Ex[Cx,Cy,Az])
        if ((n + 1) % output_step == 0) and (field_component == 'hz'):
             F[next_out, :, :] = fdtd_3d_interpolate_field(
                 f * (U + Hz[Cx, Cy,Az]), z_ind,
                 field_component)
             next_out += 1
        Hz[Cx, Cy,Az] = U

        progress = (n + 1) / Niter


    return F, t
```

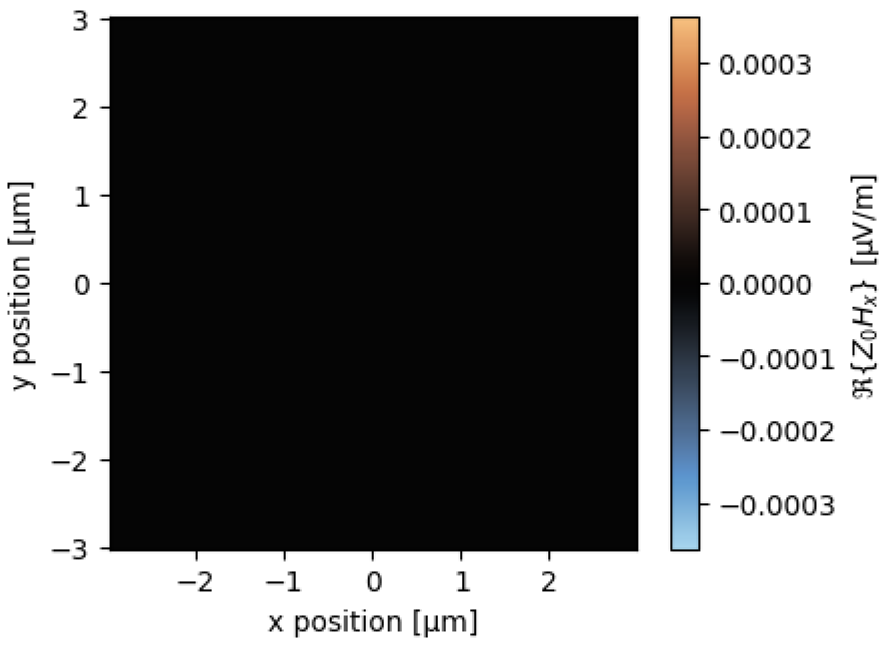
Task 2.2 : Propagation of pulse through homogeneous and inhomogeneous medium in 3D

In [ ]:
```python
#We use the test parameter here,firstly construct the plot tools
plt.rcParams.update({
    'figure.figsize': (12 / 2.54, 9 / 2.54),
    'figure.subplot.bottom': 0.15,
    'figure.subplot.left': 0.165,
    'figure.subplot.right': 0.90,
    'figure.subplot.top': 0.9,
    'axes.grid': False,
    'image.cmap': 'bluered_dark',

})
save_figures = False
# save movie to disk (requires Ffmpeg)
save_movie = False
plt.close('all')

#Setting basic parameter
c = 2.99792458e8  # speed of light [m/s]
mu0 = 4 * np.pi * 1e-7  # vacuum permeability [Vs/(Am)]
eps0 = 1 / (mu0 * c ** 2)  # vacuum permittivity [As/(Vm)]
Z0 = np.sqrt(mu0 / eps0)  # vacuum impedance [Ohm]

# grid parameter,time span and dr
Nx = 199  # nuber of grid points in x-direction
Ny = 201  # nuber of grid points in y-direction
Nz = 5  # nuber of grid points in z-direction
dr = 30e-9  # grid spacing in [m]
time_span = 10e-15  # duration of simulation [s]

# source parameters from  seminar
freq = 500e12  # pulse [Hz]
tau = 1e-15  # pulse width [s]
source_width = 2  # width of Gaussian current dist. [grid points]

#get the x,y and z number for different index,x and y from -N-1/2 to N-1/2
x = np.arange(-int(np.ceil((Nx - 1) / 2)), int(np.floor((Nx - 1) / 2)) + 1) * dr
y = np.arange(-int(np.ceil((Ny - 1) / 2)), int(np.floor((Ny - 1) / 2)) + 1) * dr
#assign the epsilon of 1 to the space
eps_rel = np.ones((Nx, Ny, Nz), dtype=np.float32)

midx = int(np.ceil((Nx - 1) / 2))
midy = int(np.ceil((Ny - 1) / 2))
midz = int(np.ceil((Nz - 1) / 2))
#source term we have in different direction
jx = np.zeros((Nx, Ny, Nz), dtype=np.float32)
jy = np.zeros((Nx, Ny, Nz), dtype=np.float32)
jz = np.tile(np.exp(-((np.arange(Nx)[:, np.newaxis, np.newaxis]- midx) / source_width) ** 2)*
# output parameters
z_ind = midz
output_step = 4

#use 3dfdtd function to simulate it
#get the hx distribution for starting hx component
field_component = 'hx'
Hx, t = fdtd_3d(eps_rel, dr, time_span, freq, tau,jx, jy, jz, field_component, z_ind, output_
#get the hy distribution for starting hy component
field_component = 'hy'
Hy, _ = fdtd_3d(eps_rel, dr, time_span, freq, tau,jx, jy, jz, field_component, z_ind, output_
#get the ez distribution for starting ez component
field_component = 'ez'
Ez, _ = fdtd_3d(eps_rel, dr, time_span, freq, tau,jx, jy, jz, field_component, z_ind, output_
#plot the field at the end of the simulation
fig = plt.figure()
# plot electric field
#Here, Ez is likely a 2D array where Ez[i, j] represents the Ez component of the electric fie
res = Ez[-1, :].T
#color_range calculates a range for coloring the plot based on the maximum absolute value of r
#pix calculates half of the pixel width based on the spacing of x values.
#extent defines the spatial extent of the plot in micrometers (xmin, xmax, ymin, ymax).
color_range = np.max(np.abs(res)) * 1e6
pix = 0.5 * (x[1] - x[0])
extent = ((x[0] - pix) * 1e6, (x[-1] + pix) * 1e6,(y[-1] + pix) * 1e6, (y[0] - pix) * 1e6)
#plt.imshow() plots the 2D array res.real * 1e6, where res.real extracts the real part of res
#vmin and vmax set the minimum and maximum values for the color scale.
plt.imshow(res.real * 1e6, vmin=-color_range, vmax=color_range,extent=extent)
cb = plt.colorbar()
plt.gca().invert_yaxis()
#R: Defines the radial positions for the polar grid. It is calculated based on the grid spacin
#p: Defines the angular positions for the polar grid, covering 360 degrees (or 2π radians) in
R = dr * np.arange(int(round(max(x) * 3 / 8 / dr)) - 1,int(round(max(x) / dr)) + 1, 2)
P = np.deg2rad(np.arange(72) * 5)
#xr and yr: Convert polar coordinates (R, phi) into Cartesian coordinates (xr, yr). These arra
xr = R[:, np.newaxis] * np.cos(P[np.newaxis, :])
yr = R[:, np.newaxis] * np.sin(P[np.newaxis, :])
#X and Y: Meshgrid of original Cartesian coordinates x and y. These grids represent the spatia
interp_points = np.hstack((xr.reshape((-1, 1)), yr.reshape((-1, 1))))
#U and V are reshaped to match the shape of xr using reshape(xr.shape). This step organizes th
U = interpn((x, y), Hx[-1, :, :].real, interp_points)
U = U.reshape(xr.shape)
V = interpn((x, y), Hy[-1, :, :].real, interp_points)
V = V.reshape(xr.shape)
#Normalization is crucial for ensuring that vector fields are represented consistently in plot
norm = np.sqrt(U ** 2 + V ** 2).max()
U /= norm
V /= norm
# plot magnetic field as vectors
plt.quiver(xr * 1e6, yr * 1e6, U, V, scale=1e-6 / dr, angles='xy',color='w', label='magnetic
l = plt.legend(frameon=False, loc='lower left')
for text in l.get_texts():
    text.set_color('w')
plt.title('$ct$ = {0:1.2f}µm'.format(t[-1] * c * 1e6))
plt.xlabel('x coordinate [µm]')
plt.ylabel('y coordinate [µm]')
cb.set_label('$\\Re\\{E_z\\}$ [µV/m]')
plt.show()
```

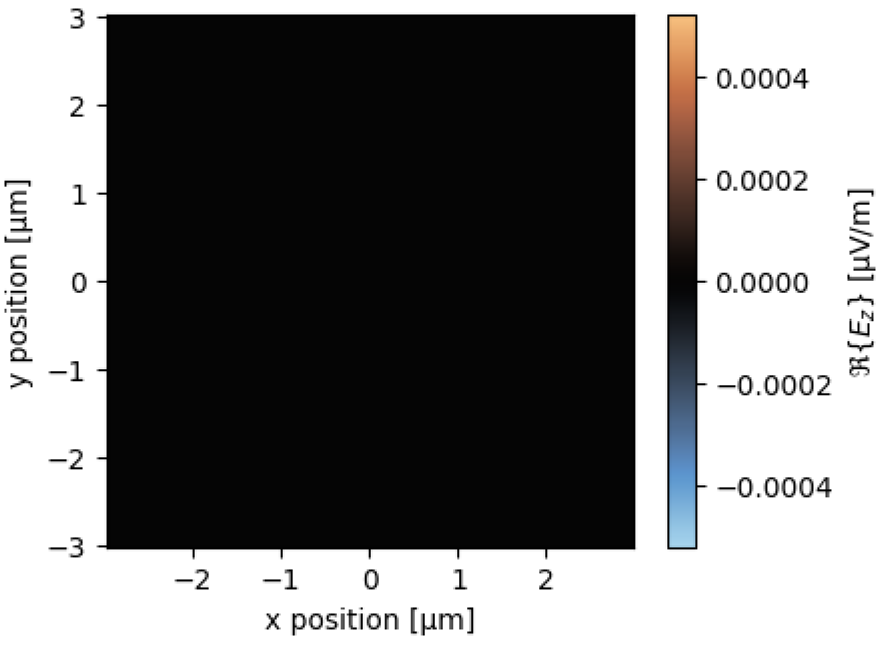


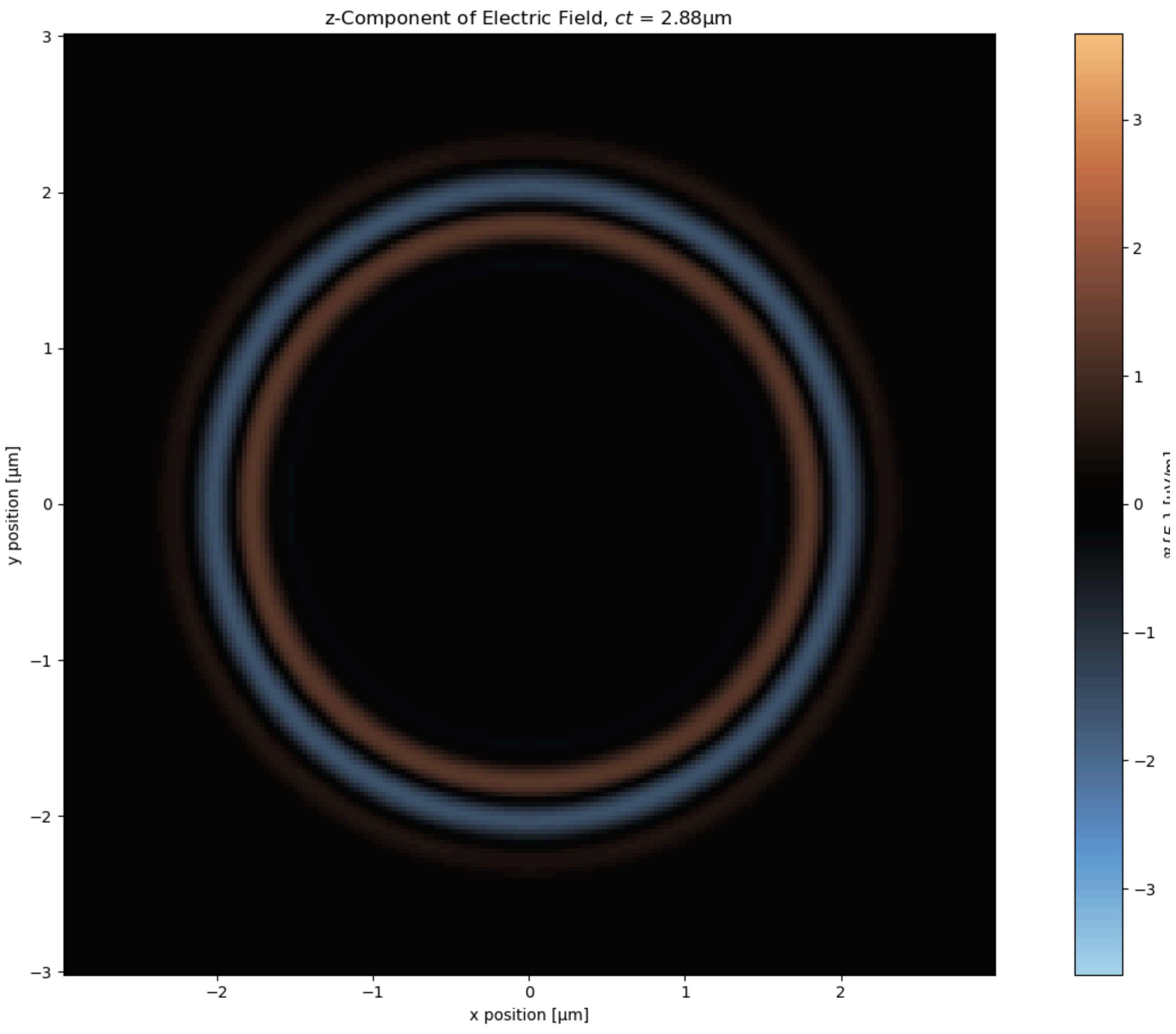
Animation for temporal evolution of electric and magnetic fields:

In [ ]:
```python
from matplotlib import animation
class Fdtd3DAnimation(animation.TimedAnimation):


    def __init__(self, x, y, t, field, titlestr, cb_label, rel_color_range, fps=25):
        # constants
        c = 2.99792458e8  # speed of light [m/s]
        self.ct = c * t

        self.fig = plt.figure()
        self.F = field
        color_range = rel_color_range * np.max(np.abs(field))
        phw = 0.5 * (x[1] - x[0])  # pixel half-width
        extent = ((x[0] - phw) * 1e6, (x[-1] + phw) * 1e6,
                  (y[-1] + phw) * 1e6, (y[0] - phw) * 1e6)
        self.mapable = plt.imshow(self.F[0, :, :].real.T,
                                  vmin=-color_range, vmax=color_range,
                                  extent=extent)
        cb = plt.colorbar(self.mapable)
        plt.gca().invert_yaxis()
        self.titlestr = titlestr
        self.text = plt.title('')
        plt.xlabel('x position [μm]')
        plt.ylabel('y position [μm]')
        cb.set_label(cb_label)
        super().__init__(self.fig, interval=1000 / fps, blit=False)

    def new_frame_seq(self):
        return iter(range(self.ct.size))

    def _init_draw(self):
        self.mapable.set_array(np.nan * self.F[0, :, :].real.T)
        self.text.set_text('')

    def _draw_frame(self, framedata):
        i = framedata
        self.mapable.set_array(self.F[i, :, :].real.T)
        self.text.set_text(self.titlestr
                           + ', $ct$ = {0:1.2f}μm'.format(self.ct[i] * 1e6))
        self._drawn_artists = [self.mapable, self.text]
# Magnetic Field Animation:

#Prepares the magnetic field data.
F = Hx * Z0 * 1e6
#Sets the title.
titlestr = 'x-Component of Magnetic Field'
#Sets the color bar label.
cb_label = '$\\Re\\{Z_0H_x\\}$ [μV/m]'
rel_color_range = 1 / 3
fps = 10
#Initializes the animation.
ani = Fdtd3DAnimation(x, y, t, F, titlestr, cb_label, rel_color_range, fps)
plt.show()

if save_movie:
    try:
        print('saving movie to disk')
        ani.save("video_FDTD3D_Hx.mp4", bitrate=1024, fps=fps, dpi=120)
    except Exception as e:
        print('saving movie failed', file=sys.stderr)
        print(e, file=sys.stderr)
#Prepares the electric field data
F = Ez * 1e6
titlestr = 'z-Component of Electric Field'
cb_label = '$\\Re\\{E_z\\}$ [μV/m]'
rel_color_range = 1 / 3
fps = 10


ani = Fdtd3DAnimation(x, y, t, F, titlestr, cb_label, rel_color_range, fps)
plt.show()
```



```
/usr/local/lib/python3.10/dist-packages/matplotlib/animation.py:884: UserWarning: Animation wa
s deleted without rendering anything. This is most likely not intended. To prevent deletion, a
ssign the Animation to a variable, e.g. `anim`, that exists until you output the Animation usi
ng `plt.show()` or `anim.save()`.
  warnings.warn(
```





Field component with time step of 2.88

Summary:

The final plot shows the distribution of the electric field component and the magnetic field components. at the end of the FDTD simulation. Here are the key features of the plot: The plot uses a color map to represent the real part of the component The color bar on the right side of the plot indicates the scale of the electric field in microvolts per meter ($\mu V/m$). The field appears to be circularly symmetric, indicating a radial distribution. Magnetic Field The magnetic field is represented by white vectors (arrows) overlaid on the electric field distribution. These vectors indicate the direction and relative magnitude of the magnetic field in the xy plane. The magnetic field vectors form concentric circular patterns around the center of the plot, indicating rotational symmetry. Plot Titles and Labels: The title indicates the simulation time corresponding to the plot: ct=3.00μm The x-axis and y-axis labels indicate positions in micrometers (μm). The legend at the bottom-left indicates that the white vectors represent the real part of the magnetic field.

Analysis:

Symmetry and Field Distribution:

The circular symmetry of both electric and magnetic fields suggests that the source or initial conditions were radially symmetric. The concentric rings of electric field intensity likely result from the propagation of electromagnetic waves originating from a central source. Field Interaction:

The pattern of the magnetic field vectors shows that the magnetic field is perpendicular to the electric field, consistent with Maxwell's equations for electromagnetic waves. The alternating directions of the electric field (color changes from blue to brown) suggest standing wave patterns, which might result from interference effects. Wave Propagation:

The circular rings suggest radial propagation of waves. The distance between rings can provide information about the wavelength of the source frequency. Field Magnitude:

The maximum and minimum values on the color bar show that the electric field varies within the range of approximately ±1.5μV/m. The normalization of the magnetic field vectors ensures that the lengths of the vectors are consistent, highlighting directional information rather than absolute magnitudes.

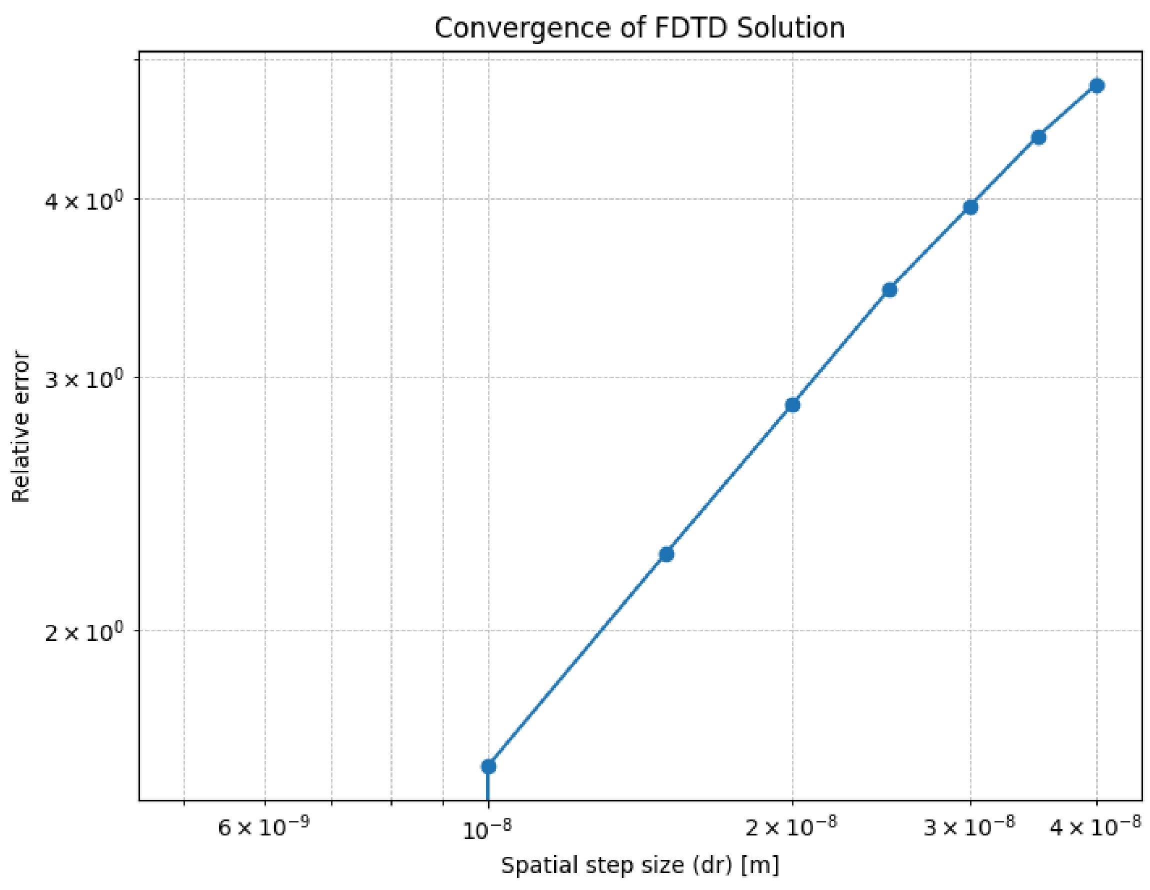
Conclusion:

The final plot effectively visualizes the electric and magnetic field components at a specific time in the simulation. The circular symmetry, radial propagation, and perpendicular orientation of the fields are consistent with theoretical expectations for an electromagnetic wave source. This result demonstrates the FDTD method's ability to accurately simulate complex field interactions and wave propagation in a 3D space.

Convergence Test of 3D Case#

In [ ]:
```python
def calculate_error(reference, result):
    return np.linalg.norm(reference - result) / np.linalg.norm(reference)

def fdtd_convergence_test(eps_rel, dr_list, time_span, freq, tau, jx, jy, jz, field_component
    c = 2.99792458e8  # speed of light [m/s]
    results = {}
    errors = []

    for dr in dr_list:
        dt = dr / (2 * c)  # Time step corresponding to the spatial step
        Nx, Ny, Nz = eps_rel.shape
        time_span = 10e-15  # duration of simulation [s]
        field, _ = fdtd_3d(eps_rel, dr, time_span, freq, tau, jx, jy, jz, field_component, z_
        results[dr] = field[-1, :, :]  # Store the last field distribution

    # Use the finest grid as the reference solution
    reference = results[dr_list[-1]]

    for dr in dr_list:
        errors.append(calculate_error(reference, results[dr]))

    return errors, dr_list

def plot_error(errors, dr_list):
    plt.figure(figsize=(8, 6))
    plt.loglog(dr_list, errors, marker='o')
    plt.xlabel('Spatial step size (dr) [m]')
    plt.ylabel('Relative error')
    plt.title('Convergence of FDTD Solution')
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)
    plt.show()

# Define the range of spatial step sizes (in meters)
dr_list = [40e-9, 35e-9, 30e-9, 25e-9, 20e-9, 15e-9, 10e-9, 5e-9]

# Run the convergence test
errors, dr_list = fdtd_convergence_test(eps_rel, dr_list, time_span, freq, tau, jx, jy, jz, f

# Plot the error versus dr
plot_error(errors, dr_list)
```

The result is:



X-axis represents the spatial step size (dr) on a logarithmic scale.Y-axis represents the relative error of the simulation results compared to the reference solution (finest grid) on a logarithmic scale. The relative error decreases as the spatial step size (dr) decreases, indicating that the solution becomes more accurate with smaller step sizes.