

Programming Computational Electromagnetic Applications Assisted by Large Language Models

Leandro Carísio Fernandes

Abstract: This article discusses the possibilities and limitations of using large language models (LLMs) in software development with applications in computational electromagnetics. Three tasks are discussed: code translation, code generation, and code description. The tests showed that LLMs are generally very useful. Even when errors occurred, they can provide useful hints to find a solution.

Index Terms: Computational electromagnetics, large language models, ChatGPT

1. Introduction

Since OpenAI launched ChatGPT on November 30, 2022, there has been an increase in interest in systems that interact with users using natural language. From the users point of view, it's a tool that seems to understand what they write, answering in natural language with coherent and correct texts. This type of tool uses language models (LMs), which are models that assign a probability to a sequence of words [1], and in general, its ultimate goal is to predict the next words given some text [2], which can be used in classification, translation, text generation, etc.

The idea of LMs has been around for some time, but a lot has changed since then. In the 1990s, this topic was addressed by statistical models that considered the n-grams model (a set of n words). At that time, there was an attempt to map words into similar classes (i.e. [3, 4]). In the 2000s, neural language models predicted the next word for a sentence using feed-forward or recurrent neural networks. During the training stage of these models, a meaningful vector representation for each word was also learned [5].

In 2017, a new sequence modeling architecture called Transformer was launched [6]. In 2018, researchers showed that a language model pre-trained on a large general-domain corpus and then fine-tuned on a target task outperformed the old models in many natural language processing tasks [7]. These two advances drove the creation, in 2018, of one of the most used transformer models today, the Generative Pretrained Transformer (GPT) [8], which was updated in 2019 (GPT-2 [9]), 2020 (GPT-3 [10]), and 2023 (GPT-4 [11]).

Until then, language model training was done using task-specific datasets, which can be challenging because it depends on large amounts of specific labeled data. In this new approach, large language models (LLMs) containing billions of parameters are generated and trained only once with generic data. Latter, they can be fine-tuned for specific tasks using fewer training data.

Using the GPT-3 model, ChatGPT allows the user to "chat" with the model. Because it is a model with 175 billion parameters trained on a huge amount of text (570 GB of compressed plaintext after filtering [10]), its text generation capability has attracted attention for solving everyday problems, including tasks in software development.

The tool has already been used to solve programming book exercises [12], which are usually simple tasks, focusing on a few aspects of the language that the reader is trying to grasp. But how can LLMs assist the development of more complex systems, as in computational electromagnetics?

In this short article, I discuss some possibilities for using this tool in three tasks: (1) translating code between programming languages; (2) writing code; and (3) describing and debugging code. By the end of this paper, I hope that the reader will understand the possibilities and current

limitations of using LLMs in these tasks.

2. Discussion

This section explores the potential of ChatGPT for software development tasks with examples in computational electromagnetics. While there are similar tools, since ChatGPT is the most well-known among them, it was used in all examples. Currently, there are two available versions, 3.5 and 4. Because the former is the most used, all tests were performed on this version.

Due to space limitations in this article, the source code has been trimmed to show only the relevant part for discussion. It was also not possible to transcribe the full interactions with ChatGPT in this space. However, both the code and the conversations were saved and are available in a public repository [13].

2.1. Translating code between different languages - converting a 3D-FDTD code from MATLAB to Python

Translating code from one programming language to another is a common task, especially when studying a new method or prototyping a solution in one language and implementing it in another. To evaluate how ChatGPT performs in code translation, a MATLAB script that calculates resonant frequencies of a PEC cavity using FDTD-3D was translated to Python. The code was created by Prof. Davidson and is freely available on his book's website [14].

This is an interesting problem as the two languages have some syntax similarities but also important differences. MATLAB and Python use, respectively, 1-based and 0-based array indexing, and despite having functions with the same name, the libraries and the way they are used might be different. The problem itself is illustrative of other methods since FDTD loops through a grid, which is also done in other computational EM methods.

Due to the size of the code, it was split into three parts. This was necessary so the text fits into the model's context window, which is 2048 tokens for GPT-3 (a token is a subunit of text, usually something between a character and a word [8, 10]). This can be a limiting factor for large codes because one snippet may be related to another that is far away in the text. Although this is a problem today, it can often be circumvented. With the advances of these models, this can quickly become irrelevant.

Listings 1 and 2 show the first part of the original and converted codes containing variable definitions. The prompt was "Convert the MATLAB code to Python: [CODE]". It can be observed that:

- The definitions of simple variables were correctly converted (e.g., `eps_0` and `N_x`).
- The return of the `input` function in MATLAB is directly assigned to the variable `refine`. In Python, the return of the function should be explicit converted from `str` to `int`, which was done in line 11.
- The argument passed to the `min` function in the definition of `Delta_s` was correctly converted from a MATLAB one-dimensional array to a Python list (a comma was inserted between the values - line 16).
- Expressions using `sqrt`, `zeros`, and `rand` functions were mapped to use the corresponding ones of the NumPy library.
- `N_t` was correctly converted, including the change of the power symbol from $^$ to `**`.

Listing 1. FDTD: Variable definitions - MATLAB code

```
1 eps_0 = 8.854e-12;  
2 ...  
3 N_x = 8;  
4 ...  
5 refine = input('Factor to refine mesh by? (default 2): ');  
6 if isempty(refine)  
7     refine = 2  
8 end
```

```

9 N_x = N_x*refine;
10 ...
11 Delta_x = L_x/N_x;
12 ...
13 Delta_s = min([Delta_x Delta_y Delta_z])
14 c = 1/sqrt(eps_0*mu_0);
15 CFL = Delta_s/(sqrt(3)*c);
16
17 N_t = 2^10*refine;
18 ...
19 E_x = zeros(N_x, N_y+1,N_z+1)
20 ...
21 H_z(:, :, :) = rand(N_x,N_y,N_z+1)-0.5;
22 ...
23 t = zeros(N_t,1);
24 t(1) = Delta_t;
25 H_z_t = zeros(N_t,1);

```

Listing 2. FDTD: Variable definitions - Code converted to Python

```

1 import numpy as np
2
3 eps_0 = 8.854e-12
4 ...
5 N_x = 8;
6 ...
7 refine = input('Factor to refine mesh by? (default 2): ')
8 if refine == '':
9     refine = 2
10 else:
11     refine = int(refine)
12 N_x = N_x*refine
13 ...
14 Delta_x = L_x/N_x;
15 ...
16 Delta_s = min([Delta_x, Delta_y, Delta_z])
17 c = 1/np.sqrt(eps_0*mu_0)
18 CFL = Delta_s/(np.sqrt(3)*c)
19
20 N_t = 2**10*refine
21 ...
22 E_x = np.zeros((N_x, N_y+1, N_z+1))
23 ...
24 H_z[:, :, :] = np.random.rand(N_x, N_y, N_z+1)-0.5
25 ...
26 t = np.zeros((N_t, 1))
27 t[0] = Delta_t
28 H_z_t = np.zeros((N_t, 1))

```

The first part of the code is simple and was easily translated without any issues. The second part is more complex and deals with the core of FDTD, i.e. a temporal loop that updates the electromagnetic fields using the finite difference method. Listings 3 and 4 show the original code in MATLAB and the converted one to Python:

Listing 3. FDTD: Main loop - MATLAB code

```

1 for n=1:N_t
2     t(n) = (n)*Delta_t;
3     H_z_t(n) = H_z(round(N_x/2), round(N_y/2), round(N_z/2));
4
5     H_x = H_x + Delta_t/mu * (diff(E_y,1,3)/Delta_z - diff(E_z,1,2)/Delta_y);
6     H_y = H_y + Delta_t/mu * (diff(E_z,1,1)/Delta_x - diff(E_x,1,3)/Delta_z);
7     H_z = H_z + Delta_t/mu * (diff(E_x,1,2)/Delta_y - diff(E_y,1,1)/Delta_x);
8

```

```

9   E_x(:,2:N_y,2:N_z) = E_x(:,2:N_y,2:N_z) + Delta_t/epsilon *
    (diff(H_z(:, :, 2:N_z), 1, 2)/Delta_y - diff(H_y(:, 2:N_y, :), 1, 3)/Delta_z);
10  E_y(2:N_x, :, 2:N_z) = E_y(2:N_x, :, 2:N_z) + Delta_t/epsilon *
    (diff(H_x(2:N_x, :, :), 1, 3)/Delta_z - diff(H_z(:, :, 2:N_z), 1, 1)/Delta_x);
11  E_z(2:N_x, 2:N_y, :) = E_z(2:N_x, 2:N_y, :) + Delta_t/epsilon *
    (diff(H_y(:, 2:N_y, :), 1, 1)/Delta_x - diff(H_x(2:N_x, :, :), 1, 2)/Delta_y);
12 end

```

Listing 4. FDTD: Main loop - Code converted to Python

```

1  for n in range(1, N_t):
2      t[n] = n*Delta_t
3      H_z_t[n] = H_z[int(N_x/2), int(N_y/2), int(N_z/2)]
4
5      H_x = H_x + Delta_t/mu * (np.diff(E_y,axis=2)/Delta_z -
    np.diff(E_z,axis=1)/Delta_y)
6      H_y = H_y + Delta_t/mu * (np.diff(E_z,axis=0)/Delta_x -
    np.diff(E_x,axis=2)/Delta_z)
7      H_z = H_z + Delta_t/mu * (np.diff(E_x,axis=1)/Delta_y -
    np.diff(E_y,axis=0)/Delta_x)
8
9      E_x[1:N_x,1:N_y+1,1:N_z+1] = E_x[1:N_x,1:N_y+1,1:N_z+1] + Delta_t/epsilon *
    (np.diff(H_z,axis=2)/Delta_y - np.diff(H_y,axis=1)/Delta_z)
10     E_y[1:N_x+1,1:N_y,1:N_z+1] = E_y[1:N_x+1,1:N_y,1:N_z+1] + Delta_t/epsilon *
    (np.diff(H_x,axis=2)/Delta_z - np.diff(H_z,axis=0)/Delta_x)
11     E_z[1:N_x+1,1:N_y+1,1:N_z] = E_z[1:N_x+1,1:N_y+1,1:N_z] + Delta_t/epsilon *
    (np.diff(H_y,axis=0)/Delta_x - np.diff(H_x,axis=1)/Delta_y)

```

At first glance, it seems to be a successful conversion. However, there are some problems with the converted code, primarily involving how both languages work with indexes:

- There is a semantic error in line 1. The MATLAB code loops through the interval `1:N_t`, which represents the interval `[1, N_t]`. The Python code uses `range(1, N_t)`, which represents the interval `[1, N_t)`. Since in MATLAB indexing starts at 1, this interval should have been converted to `[0, N_t)`.
- The access to the indexes of `H_z_t` was incorrectly translated (line 3). In MATLAB, `round` rounds to the nearest integer. In Python, `int` truncates a `float` passed as a parameter, rounding it down. The indexing issue was also not addressed.
- The calculation of the magnetic fields (lines 5-7) was correctly converted. The function `diff` was translated to `np.diff` and its parameters values were handled (axes 1, 2, 3 were adapted to 0, 1, 2).
- The arguments of the `diff` function was correctly handled in the definition of `E_y` and `E_z` (lines 10-11). However, this does not occur for `E_x`: the `axis` parameter should be changed from (2, 3) to (1, 2), but it was changed to (2, 1) (line 9).
- The slicing of the matrices in lines 9-11 was incorrectly converted to Python. For example, the original slicing of `E_x` was `(:, 2:N_y, 2:N_z)` and it was adapted to `[1:N_x, 1:N_y + 1, 1:N_z + 1]`, but it should have been converted to `[:, 1:N_y, 1:N_z]`.

Finally, the third part of the code calculates the FFT of `H_z` and plots the resonant frequencies. Listings 5 and 6 show the original and converted codes. We can observe that:

- `Delta_f` (line 1) was translated to `delta_f`, with lowercase d (line 1).
- The loop through `freq` faces the same indexing problem as the main FDTD loop, i.e. it is shifted by one unit.
- The function `fft` was converted to `np.fft.fft`. However, the equivalent code should be `np.fft.fft(H_z_t, axis=0)`. By not setting `axis=0`, the function uses its default value (-1).
- The plotting code was converted correctly.

Listing 5. FDTD: FFT and plot - MATLAB code

```

1 Delta_f = 1/(N_t*Delta_t);
2 for kk = 1:N_t/2
3     freq(kk) = (kk-1)*Delta_f;
4 end
5
6 k_exact = [5.2359878 7.55154489 8.1788743 8.9472598];
7 f_exact = k_exact * c/(2*pi);
8 H_z_f = fft(H_z_t);
9
10 dummy = [0 0 0 0];
11 plot(freq,abs(H_z_f(1:N_t/2)),f_exact,dummy,'x');
12
13 figure
14 end_pt = floor(5e8/Delta_f)
15 norm = max(abs(H_z_f(2:end_pt)))
16 plot(freq(2:end_pt)/1e6,abs(H_z_f(2:end_pt))/norm)
17 f_exact = f_exact/1e6;
18 line([f_exact(1) f_exact(1)],[0 1],'LineStyle','--','Color','k')
19 line([f_exact(2) f_exact(2)],[0 1],'LineStyle','--','Color','k')
20 line([f_exact(3) f_exact(3)],[0 1],'LineStyle','--','Color','k')
21 line([f_exact(4) f_exact(4)],[0 1],'LineStyle','--','Color','k')
22 xlabel('f (MHz)')
23 ylabel('H_z (normalized)')

```

Listing 6. FDTD: FFT and plot - Code converted to Python

```

1 delta_f = 1 / (N_t * delta_t)
2 freq = np.zeros(int(N_t/2))
3
4 for kk in range(1, int(N_t/2)+1):
5     freq[kk-1] = (kk-1) * delta_f
6
7 k_exact = [5.2359878, 7.55154489, 8.1788743, 8.9472598]
8 f_exact = [k*c/(2*np.pi) for k in k_exact]
9 H_z_f = np.fft.fft(H_z_t)
10
11 dummy = np.array([0, 0, 0, 0])
12 plt.plot(freq, np.abs(H_z_f[0:N_t//2]), f_exact, dummy, 'x')
13 plt.pause(0.01)
14
15 fig = plt.figure()
16 end_pt = int(5e8/Delta_f)
17 norm = max(np.abs(H_z_f[1:end_pt]))
18 plt.plot(freq[1:end_pt]/1e6, np.abs(H_z_f[1:end_pt])/norm)
19 f_exact = [f/1e6 for f in f_exact]
20 plt.plot([f_exact[0], f_exact[0]], [0, 1], linestyle='--', color='k')
21 plt.plot([f_exact[1], f_exact[1]], [0, 1], linestyle='--', color='k')
22 plt.plot([f_exact[2], f_exact[2]], [0, 1], linestyle='--', color='k')
23 plt.plot([f_exact[3], f_exact[3]], [0, 1], linestyle='--', color='k')
24 plt.xlabel('f (MHz)')
25 plt.ylabel('H_z (normalized)')

```

2.2. Writing code

The essence of software development is writing code. In a new software, a significant part of the code is built from algorithms or snippets that have already been written by someone else and that can be adapted to suit the new software. An LLM can assist in this process. Although it is still not possible to generate fully functional complex software without any intervention, the tool can be used to write codes for common tasks.

2.2.1. Neural network for propagation problems

Let's consider the calculation of path loss at 900 MHz in urban environments when the transmitting antenna is below the building level. This problem has already been addressed in [15] and here we are only interested in the code generated by the LLM. In [15], the path loss was calculated with 2 inputs that represent the environment between transmitter and receiver: the distance between them and a numerical information about the amount of area occupied by buildings.

The following prompt was used to build a neural network to solve the same problem: *"Give me a Python code using scikit-learn that trains a multi-layer perceptron. The input is a NumPy array with two columns, each one representing one variable. The output should be a numerical value (it's a regression problem). Normalize the inputs."* Listing 7 shows the generated code:

Listing 7. Python code generated by ChatGPT used to train a simple neural network

```
1 import numpy as np
2 from sklearn.neural_network import MLPRegressor
3 from sklearn.preprocessing import StandardScaler
4
5 # generate some example data
6 X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
7 y = np.array([10, 20, 30, 40])
8
9 # normalize the inputs
10 scaler = StandardScaler()
11 X_normalized = scaler.fit_transform(X)
12
13 # create an MLPRegressor with one hidden layer
14 mlp = MLPRegressor(hidden_layer_sizes=(100,), max_iter=1000, alpha=0.01)
15
16 # train the MLPRegressor on the normalized inputs
17 mlp.fit(X_normalized, y)
18
19 # make predictions on new inputs
20 X_new = np.array([[2, 3], [4, 5]])
21 X_new_normalized = scaler.transform(X_new)
22 y_pred = mlp.predict(X_new_normalized)
23
24 print(y_pred)
```

The code is clear and includes comments that allow easy identification of what needs to be modified. Figure 1 shows the path loss calculated after plugging in the data from [15]. We can visually compare the results and see that, even without any optimization, the code is quite useful.

2.2.2. Method of moments for calculating the radiation pattern of a dipole using the method of moments

In this example, I requested a code to calculate the radiation pattern of a dipole antenna, which is useful to test the generation of a more complex code. The prompt used was *"Write a code to calculate the radiation pattern of a dipole antenna in Python using the method of moments"*. The output is shown in listing 8.

Listing 8. MoM code to plot the radiation pattern of a dipole

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # Define dipole geometry
5 L = 0.5      # length in meters
6 d = 0.1      # distance from reference point in meters
7
8 # Define number of elements and element length
9 nelelem = 20
10 lelem = L / nelelem
11
```

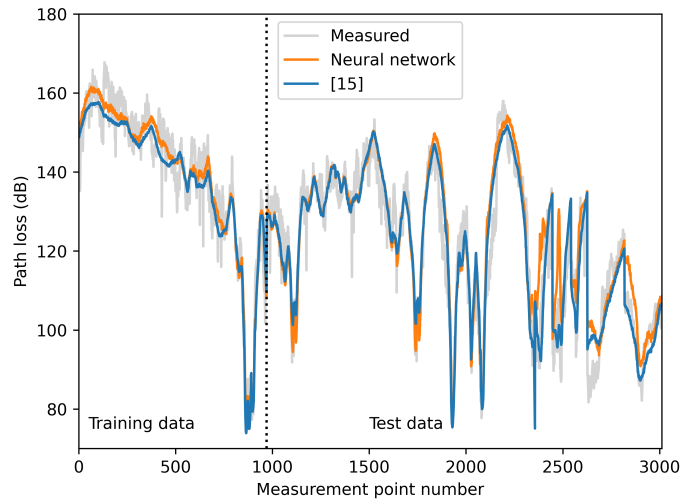


Fig. 1. Measured and predicted path loss

```

12 # Define impedance matrix
13 Z = np.zeros((nelem, nelem), dtype=np.complex128)
14
15 # Calculate mutual impedances
16 for i in range(nelem):
17     for j in range(nelem):
18         if i == j:
19             Z[i,j] = 0.5j*np.pi
20         else:
21             r = np.sqrt((i-j)**2*lelem**2 + d**2)
22             Z[i,j] = np.exp(-1j*2*np.pi*r/0.3)*1j/4/np.pi/r
23
24 # Define voltage vector
25 V = np.zeros((nelem,1), dtype=np.complex128)
26 V[0] = 1.
27
28 # Solve for current vector
29 I = np.linalg.solve(Z, V)
30
31 # Calculate radiated field in x-y plane
32 theta = np.linspace(0, np.pi, 361)
33 Etheta = np.zeros_like(theta)
34 for th in range(len(theta)):
35     for i in range(nelem):
36         r = np.sqrt(i**2*lelem**2 + d**2)
37         Et = np.exp(-1j*2*np.pi*r/0.3)*I[i]*np.sin(theta[th])*lelem
38         Etheta[th] += Et
39
40 # Plot radiation pattern
41 plt.polar(theta, np.abs(Etheta))
42 plt.title("Radiation Pattern of Half-Wave Dipole Antenna")
43 plt.show()

```

While running this code, an error was reported on line 38. This is because `Et` is a `np.array` of length 1 and `Etheta[th]` is a `float`. When informed about the error, ChatGPT suggested changing to `Etheta[th] = Etheta[th] + Et` (i.e. just changing the operator `+=`), so the error persisted.

Another attempt was made to be more explicit by using the prompt: "It does not solve the problem, an error is still occurring: `ValueError: setting an array element with a sequence. Et is an`

array of one element. Shouldn't you extract the first element of the array?". This time, ChatGPT suggested to change the code to `Etheta[th] += Et[0]`, which allowed it to be executed, and figure 2 was displayed.

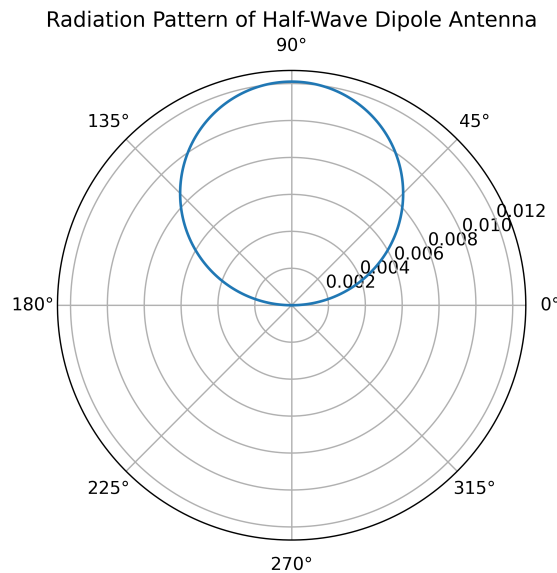


Fig. 2. Radiation pattern of a dipole as generated by ChatGPT

2.3. Code description and debugging

Listing 9 shows a code snippet from EM Studio [FDTD] that handles the electric field update around a rectangular PEC on a 3D grid using the FDTD method [16]. The code was inputted into ChatGPT with the prompt "What is the purpose of this code?", and it answered as shown in listing 10. Even without the complete code of the whole program, it provided the correct explanation and made good assumptions that are not explicit in the code.

Listing 9. Code snippet to update the electric field around a rectangular PEC

```

1 void RectangularPECSheetFDTDCore::ElectricFieldsChanged(double t) {
2     FDTDEngine* engine = static_cast<FDTDEngine*>(GetEngine());
3
4     if (dto.IsCutAtZ()) {
5         for (int i = imin; i < imax; i++) {
6             for (int j = jmin; j <= jmax; j++) {
7                 engine->SetEx(i, j, kmin, 0);
8             }
9         }
10        for (int i = imin; i <= imax; i++) {
11            for (int j = jmin; j < jmax; j++) {
12                engine->SetEy(i, j, kmin, 0);
13            }
14        }
15    }
16    ...
17 }

```

Listing 10. ChatGPT explaining the code from listing 9

This code is a method implementation of a class called RectangularPECSheetFDTDCore. It is part of a larger program that likely performs Finite-Difference Time-Domain (FDTD) simulations for electromagnetic field propagation in a rectangular region.

The purpose of this specific method `ElectricFieldsChanged` is to set the electric field components at certain locations to zero. This is done differently depending on whether the simulation domain is cut along the x, y or z direction, as specified by the `dto` object.

For instance, if `dto.IsCutAtZ()` is true, then the x- and y-components of the electric field at the minimum z-plane are set to zero. Similarly, if `dto.IsCutAtX()` is true, then the y- and z-components of the electric field at the minimum x-plane are set to zero. And if `dto.IsCutAtY()` is true, then the x- and z-components of the electric field at the minimum y-plane are set to zero.

This method is likely called at the beginning of each time step of the simulation to ensure that the electric fields satisfy any boundary conditions that may have been imposed.

Listing 11 shows the FDTD loop for calculating the electric field in a 3D grid [16]. A multiplication sign was removed from line 9. The code was passed to ChatGPT along with the prompt *"What is the purpose of this code? Is there something wrong with it?"*, and it replied as shown in listing 12.

Listing 11. Main loop to calculate the electric field in a general purpose FDTD software

```
1 void FDTDData::CalculateEFields(Mesh& mesh) {
2     int i, j, k;
3     for(i = 0; i < NxTOTAL; i++) {
4         for(j = 0; j < NyTOTAL; j++) {
5             for(k = 0; k < NzTOTAL; k++) {
6                 MaterialCoefficients mc = mesh.GetCoefficients(i, j, k);
7                 Ex[i][j][k] = mc.GetGax()*(Dx[i][j][k] - ix[i][j][k]);
8                 ix[i][j][k] = ix[i][j][k] + mc.GetGbx()*Ex[i][j][k];
9                 Ey[i][j][k] = mc.GetGay() (Dy[i][j][k] - iy[i][j][k]);
10                iy[i][j][k] = iy[i][j][k] + mc.GetGby()*Ey[i][j][k];
11                Ez[i][j][k] = mc.GetGaz()*(Dz[i][j][k] - iz[i][j][k]);
12                iz[i][j][k] = iz[i][j][k] + mc.GetGbz()*Ez[i][j][k];
13            }
14        }
15    }
16 }
```

Listing 12. ChatGPT explaining the code from listing 11

The purpose of this code is to calculate the electric field values at every point in a given three-dimensional mesh using the finite-difference time-domain (FDTD) method. It updates the Ex, Ey, and Ez arrays, which represent the electric field components in the x, y, and z directions, respectively, at every point (i, j, k) in the mesh.

The code loops through every point in the mesh using three nested loops, and for each point, it calculates the electric field values by applying a set of equations involving the material coefficients and the values of the electric field and the time derivative of the electric field at that point and its neighboring points.

Without more context, it is difficult to determine whether there is anything wrong with the code. It may depend on the implementation of the Mesh class and the MaterialCoefficients class, as well as how the Dx, Dy, and Dz arrays are initialized and updated elsewhere in the code. Additionally, the code does not include any error checking or bounds checking, so it is possible that it could result in errors or exceptions if the input values are not valid.

The first answer did not identify the error. However, ChatGPT provided some potential sources of problems. When the error returned by the compiler (error: expression cannot be used as a function)

was informed, it suggested correct it as `Ey[i][j][k] = mc.GetGay * (Dy[i][j][k] - iy[i][j][k])`, treating `GetGay` as a member variable. Without any context, this could be a possible correction. However, any human would interpret that `GetGay` is a method, not a member variable. By explicitly informing this to ChatGPT, it was able to correct the error to `Ey[i][j][k] = mc.GetGay() * (Dy[i][j][k] - iy[i][j][k])`.

3. Conclusions

This article showed the use of large language models for tasks related to software development with applications in computational electromagnetics. Three types of tasks were analyzed using ChatGPT.

The task of converting MATLAB FDTD code to Python was performed without any issues for simple variable declarations and assignments. Conversion of functions calls was satisfactory, but with some minor problems, particularly in passing parameters to `np.fft.fft`. However, there were difficulties converting code that involves matrix indexing, likely due to differences in array indexing starting at 0 in Python and 1 in the MATLAB.

The code generation task presented two examples. The code to train a neural network to predict path loss was successful on the first attempt. The code to calculate the radiation pattern of a dipole using the method of moments failed to compile but, after some instructions, the error was corrected and the radiation pattern was presented. Code generation proved to be a curious task. In the first example, clean and easy-to-understand code was generated. In the second, many things were not clear. For example, at which frequency was the diagram calculated? Is the code correct? Blindly trusting the result is not possible and a deeper analysis of the code is required.

The task of describing and debugging code was successful, as ChatGPT was able to generate explanations for small code snippets and even provide insights beyond what was explicitly written in the code. Using LLM for debugging proved to be interesting, as it provides more understandable messages compared to traditional compiler errors, although some suggested corrections may not always be accurate.

Besides the evaluated tasks, language models can also assist in simpler software development tasks. For instance, when a programmer cannot remember the name of a library, they can give vague guidelines to the LLM, which can provide not only the correct library but also a basic code structure to adapt for personal use, saving time and making the development process more efficient.

It is necessary to consider that the goal of an LLM is to predict the next word given a context. While from the user's point of view the model is answering questions or performing some tasks, what is being done is a statistical treatment of the input data sequence. Although this implies that eventually the generated text may be completely wrong, it is impressive what is obtained with these models. In practice, even when errors are provided, the model can provide useful hints for the user and could certainly assist software development.

References

- [1] Daniel Jurafsky, and James H. Martin, *Speech and Language Processing - An introduction to natural language processing, computational linguistics, and speech recognition*, 3rd ed. Cambridge University Press, 2023. [Online]. Available: <https://web.stanford.edu/~jurafsky/slp3/>
- [2] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie and Ji-Rong Wen. (2023) A survey of Large Language Models. [Online]. Available: <https://arxiv.org/pdf/2303.18223.pdf>
- [3] P. F. Brown, V. J. Della Pietra, P. V. deSouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational Linguistics*, vol. 18, pp. 467–479, 1992.
- [4] Fernando Pereira and Naftali Tishby and Lillian Lee, "Distributional Clustering Of English Words," in *Proceedings of the ACL*, 1993, pp. 183–190.

- [5] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of Machine Learning Research*, vol. 3, pp. 1137–1155, 2003.
- [6] Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin, "Attention Is All You Need," 2017.
- [7] Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-tuning for Text Classification," 2018.
- [8] Lewis Tunstall, Leandro von Werra, and Thomas Wolf, *Natural Language Processing with Transformers - Building Language Applications with Hugging Face*, 1st ed. O'Reilly, 2022.
- [9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," 2019. [Online]. Available: <https://www.semanticscholar.org/paper/Language-Models-are-Unsupervised-Multitask-Learners-Radford-Wu/9405cc0d6169988371b2755e573cc28650d14dfe>
- [10] Tom B. Brown and Benjamin Mann and Nick Ryder and Melanie Subbiah and Jared Kaplan and Prafulla Dhariwal and Arvind Neelakantan and Pranav Shyam and Girish Sastry and Amanda Askell and Sandhini Agarwal and Ariel Herbert-Voss and Gretchen Krueger and Tom Henighan and Rewon Child and Aditya Ramesh and Daniel M. Ziegler and Jeffrey Wu and Clemens Winter and Christopher Hesse and Mark Chen and Eric Sigler and Mateusz Litwin and Scott Gray and Benjamin Chess and Jack Clark and Christopher Berner and Sam McCandlish and Alec Radford and Ilya Sutskever and Dario Amodei, "Language Models are Few-Shot Learners," 2020.
- [11] OpenAI, "GPT-4 Technical Report," 2023.
- [12] Allen Downey. (2023) LLM-Assisted Programming. [Online]. Available: <https://www.allendowney.com/blog/2023/04/02/llm-assisted-programming/>
- [13] Leandro Carísio Fernandes, "Tests with LLM in computational electromagnetics." [Online]. Available: https://github.com/carisio/tests_llm_cem
- [14] David B. Davidson, *Computation Electromagnetics for RF and Microwave Engineering*, 2nd ed. Cambridge University Press, 2011. [Online]. Available: <http://www.cambridge.org/Davidson>
- [15] Leandro Carísio Fernandes and Antonio José Martins Soares, "Path loss prediction in microcellular environments at 900MHz," *AEU - International Journal of Electronics and Communications*, vol. 68, no. 10, pp. 983–989, 2014.
- [16] —, "Software Architecture for the Design of Electromagnetic Simulators [EM Programmer's Notebook]," *IEEE Antennas and Propagation Magazine*, vol. 55, no. 1, pp. 155–168, 2013.