Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

School of Electronic Engineering

CB54: Machine Learning Algorithms for EM Wave Scattering Problems

Appendix D: Project Design & Implementation

Anthony James McElwee

ID Number: 20211330

August 2023

MEng in Electronic and Computer Engineering

Supervised by Dr Conor Brennan

Contents

Project Design & Implementation

Code Validation

The Python code was validated in the same manner as the MATLAB code was validated in [1]. This code is called using the function EmsctCircle in the file custom_functions_EM.py found in the folder lib found in the project GitHub repository "https://github.com/spookworm/CB54". Details of the validation code and lengthy derivation are covered in Section 3.A.1.3 of [1] and the main points with plots are briefly described below. In the code ForwardBICGSTABFFTwE.py code, by changing the variable "validation" to equal "'True'" the validation code will be reproduced. Ensure that the other two variables "guess_validation_answer" and "guess_model" are both set to 'False'.

- The parameters of the simulations are loaded using the initEM function with the custom "bessel" argument enabled to indicate that the programme returns the base scatterer geometry.

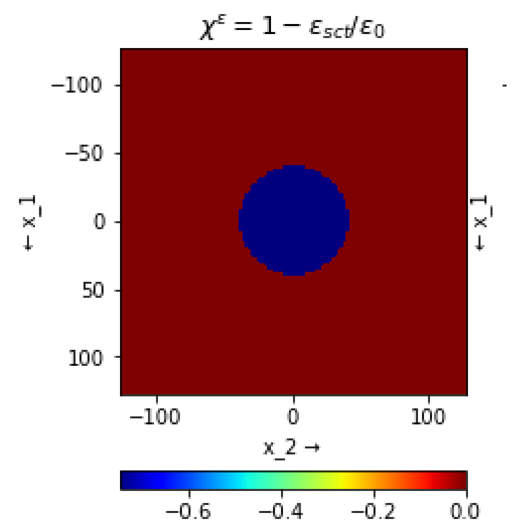


Figure 1. Validation scatterer geometry for validation case. As there is no permeability contrast.

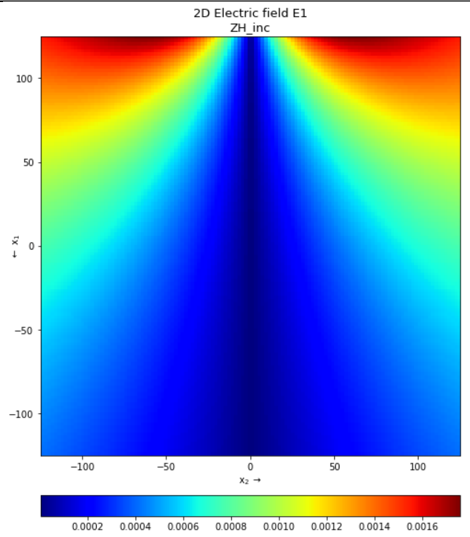


Figure 2. Incident ZH_z field.

- The receiver and source emitter have their co-ordinates transformed from cartesian to polar form.

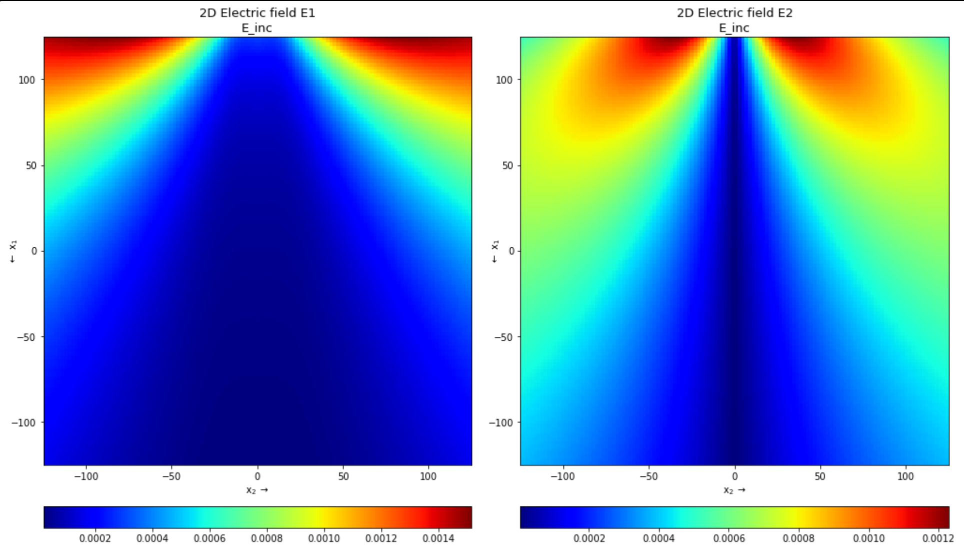


Figure 3. Incident *x* and *y* components of the incident Electric Fields.

- The Bessel-Function series are calculated in a manner to reduce the required algebraic steps using a for loop with a higher number of terms originally found in [1] due to experimentation with initEM parameters during the development process. Coefficients A and B are used in determining the reflected electric field at the receivers as denoted in equations (3.A.15-16) in [1].
- The reflected electric field and incident fields are transformed from polar co-ordinates back into cartesian co-ordinates.

- The absolute difference between the magnitudes of the analytical Bessel-Function Approach and the fields produced by the Contrast-Integral Approach is calculated and printed to the plotted diagrams below.

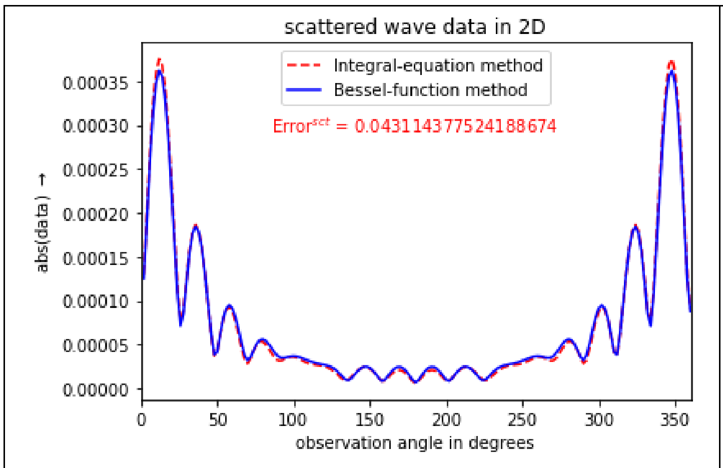


Figure 4. Scattered E-Field at receivers.

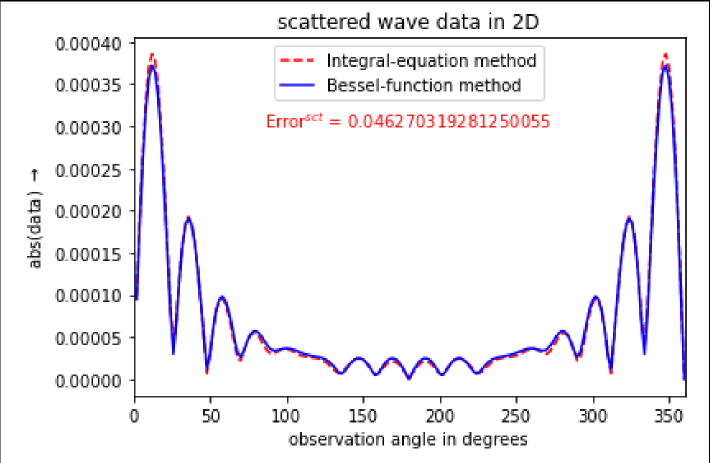


Figure 5. Scattered ZH-Field at receivers.

- The validation used in [1] is that of a 2D electric dipole line source so that comparison can be made more easily with the 3D case. Aside from validating the solver against an analytical solution, by reproducing this MATLAB code in Python, the code itself is validated for the input parameters.

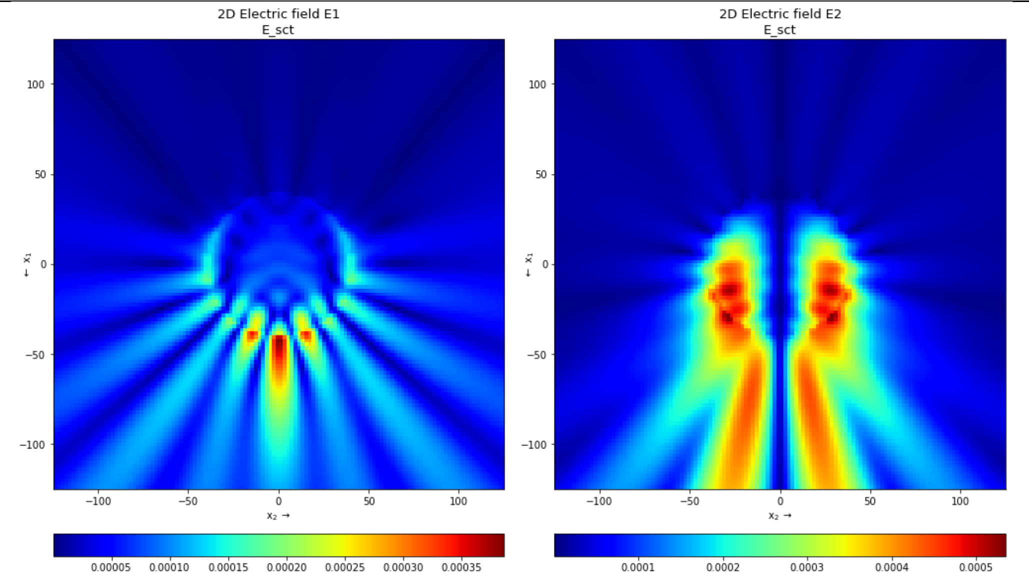


Figure 6. Final scattered electric fields for the validation case.

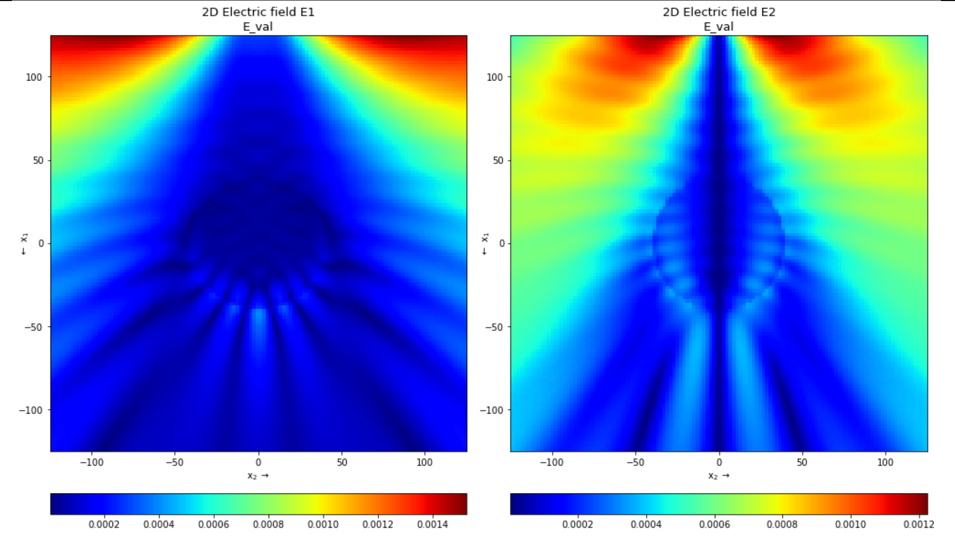


Figure 7. Final total electric fields for the validation case.

In coding the scalar code problem in Chapter 1 of [1], the student used the Python library “fn_graph” to create an illustration of the complexity of the code. While the scalar problem described is simpler the problem tackled in this project, the code flow is included below for the Bessel-Approach and the full integral solver approach.

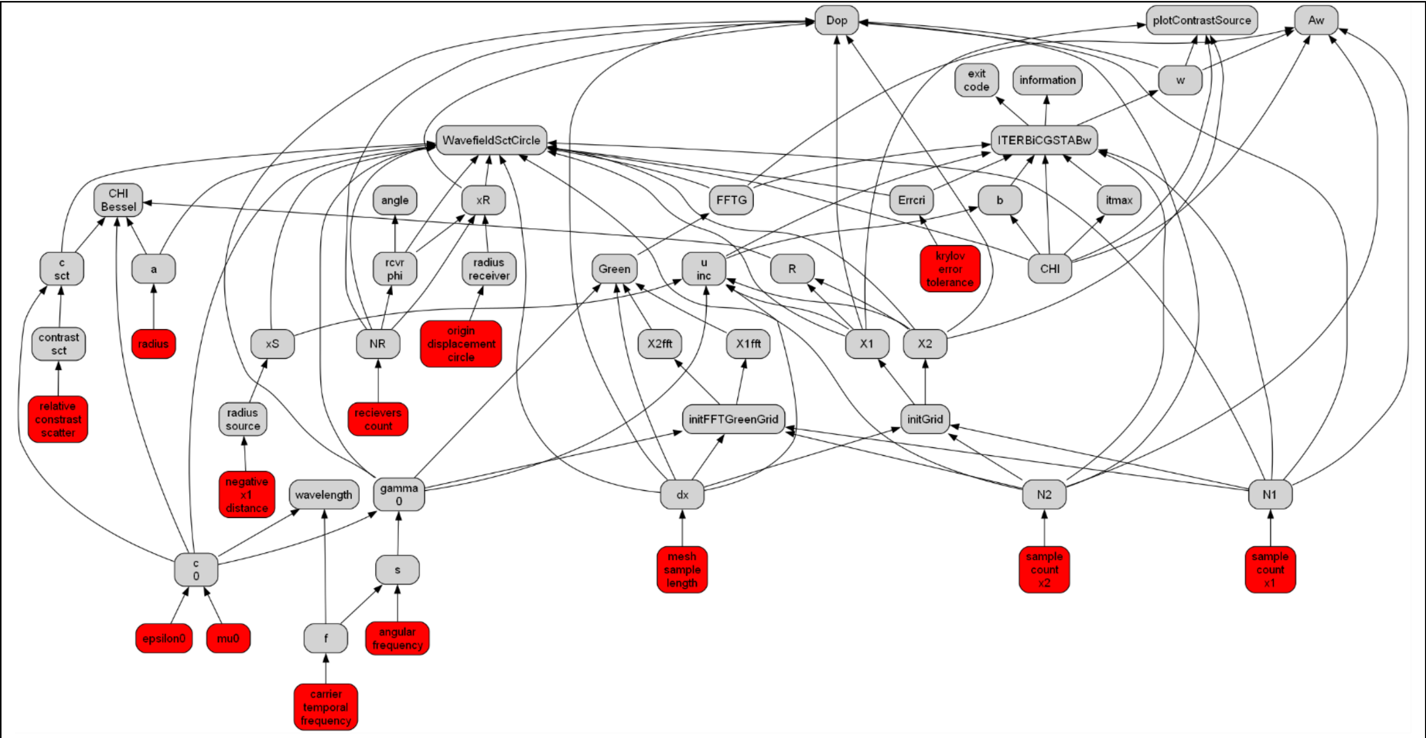


Figure 8. Bessel-Approach function flow for the scalar problem. Red indicates user inputs that determine the simulation scenario.

Model Architecture Description

The deep learning model was built using the TensorFlow and Keras libraries in Python. Details of the installation versions are found in the conda environment files on the project GitHub repository at https://github.com/spookworm/CB54 .

A simple visualisation of the layer dimensions is captured using the visualkeras Python library below. This diagram does not feature the skip connections between the encoder and decoder sides of the model.

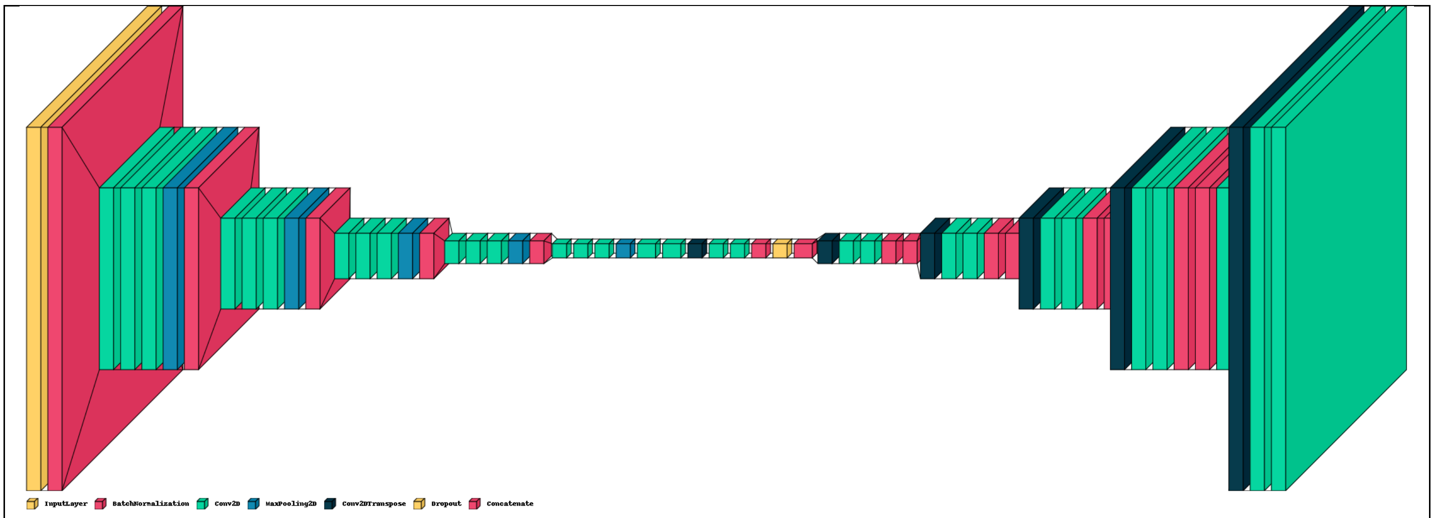
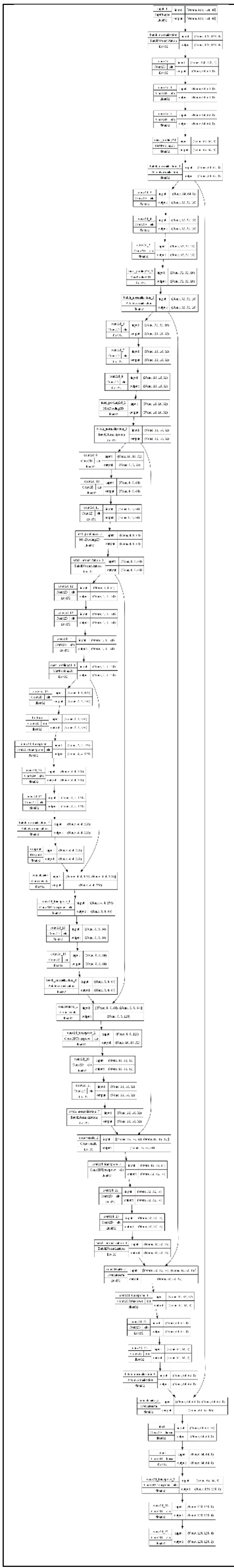


Figure 9. The model architecture used to predict both E1 and E2 scattered fields.

The full detailed model architecture is visualised using the TensorFlow plot_model library below. This can be used, aside from the code itself, to reconstruct the model architecture in future experimentation. The student recognises that the image resolution suffers within the pdf format and recommends that the reader refer directly to the source “model_plot_EM.png” file located in the subfolder “doc\Project_Design_Implementation” in the GitHub repository.



The initial input shape for the model was the batch size of the data followed by the height and width of the array with the channels given in the last position. This required transformation of the NumPy arrays the the time of training preparation using the function prescient2DL_data. The first channel consisted of the real component of the scatter geometry. If complex dielectric materials were to be used this would have to be adjusted and a second channel to hold the imaginary component of the scatterer would need to be included in the input. The rest of the channels consisted of the real, imaginary and absolute components of the relevant incident electric fields in the required direction. No information regarding the ZH field was used as this was considered constant for both fields where the carrier frequency was a constant. If the carrier frequency was to vary, the inclusion of the ZH incident wave would add information that may help in the training of the model.

The output of the model is a 128x128x2 tensor where the channels are the real and imaginary component of the scattered electric fields in the pertenaint directions. The student had coded, with code remaining in the files commented out, and considered using the custom loss functions for the Helmholtz-Hodge decomposition as well as training on a target field where just the absolute values of the fields were considered, however, time limitations and the usefulness of predicting this information to the SolverEMF2 workflow led to their deprioritisation.

The architecture used for both sub-models of Prescient2DL is called “DL_model” in the code for ease of adjustment. Other model architectures and variations remain as relics in the custom_architectures_EM.py for future reference and inspiration. The model is a U-net style architecture that uses “Elu” activation functions at every convolution and transposed convolution layer, as suggested in [2]. All layers also include a bias term that can be learned, rather than explicitly included, as in [3].

Each stage of the encoder/decoder configuration has a batch normalisation layer. Combined with the properties of the ‘Elu’ activation function, these layers reduce the probability of vanishing gradients during training [4]. Although this layer should also help the model with standardisation of inputs, the student found that it was nessecary to perform pre-processing standardisation of the inputs to ensure the model begins learning straigt-away during training. Since the standardisation proceedure can be estimated from a small batch of sample solutions, the student preferred to employ this method rather than rely purely on the model to adapt over time. The use of batch normalisation layers also makes the model more rebust to weight initialisation issues [4]. As a result, the student did not employ weight initialisation parameters in the layers as they may be prone to bad/good seeding starting points, thus moving the model development into the “seed hacking” zone. Each batch normalisation layer will also add some noise to the model, thus providing some regularization.

On the encoder side, convolution layers with stride equal to two step the height/width dimension down and increase the channel count. This is followed by a max pooling layer with pool size of 1. This essentially has no impact on the model performance as a minimum pool size is required to share information in local regions. The student left the layer in as a reminder of what the original U-net architecture employs when developing segmentation models. After the convolutions, the batch normalisation layer is again employed to skip connect layers from the encoder to the decoder side.

This encoding is carried out until a bottleneck layer is reached where the tensor dimensions are 2x2x256. A seeded dropout layer with a small value is included after the first stage of upscaling on the decoder side to add regularisation to the model.

The upscaling operations are achieved by using transposed convolution layers with stride 2. Upsampling layers were also tried but this led to stronger grid-like lines on the output predictions. Transposed convolution layers also have more trainable parameters with increases the model capacity for complexity.

At the penultimate stage of the decoder, two linear convolution layers of kernal size 3 are included in the model. Since there is a certain smoothness to the predicted fields, they are included with the aim of adding some blurring effect to the output. The student expects that this reduces some of the speckled noise that appears in the predicted versus final solution comparison graphs that could cause a sharp deteriation in the performance of the prediction as an initial guess in the SolverEMF2 workflow.

SovlerEMF2 Infusion

The manner in which Prescient2DL can be used as a stand-alone emulator is obvious: the model is provided with similar information available to the Krylov Iterative Solver and produces a prediction for the target fields with a comparable or lower error that the Krylov solver provides after one iteration. Infusing the predictions into a Method-of-Moments workflow is a more open question. The student decided to go with the simplest approach of using the predicted outputs from Prescient2DL as the initial guess for the Krylov solver. There may be more complex ways to infuse deep learning into conventional methods and the reader is referred to the recent [5] for inspiration for more entry points. As the Krylov Iterative solver at the heart of SolverEMF2 is actually searching for a solution to the contrast source as opposed to the the direct scattered fields, the output from the Prescient2DL models undergoes a transformation in ForwardBICGSTABFFTwE.py using the custom_functions_EM.KopE function so it can be used as an initial guess in the BICGSTAB solver. This is not evident in [1] since the code was not intended to be used to infer guesses. No initial guess was postulated in [1] so for the naïve implementation, the incident wave was taken to be the total wave.

SovlerEMF2 Krylov Solver

In the more explicit implementation of the MATLAB code in [1], the main solver is the Congugate Gradient Krylov solver. However, in section 3.2.3.1 it is made clear that the BICGSTAB Kyrlov solver performs much better in terms of reduced iteration count. Translating the MATLAB code to Python for these solvers was not straight forward as custom callback functions had to be written so that the outputs from the MATLAB code were comparable to the outputs produced by the ITERBiCGSTABwE function. The Python library scipy.sparse.linalg was used to provided the core bicgstab and LinearOperator functions, however, they calculate the iteration steps in a different manner to the MATLAB calculation and are not easily accesible to debug. The student has formulated the code so that the outputs from the Python code match those produced by MATLAB in case development is required due to errata in [1] or other solvers or adaptations are desired in future work.

Bibliography

[1] P. M. van den Berg, *Forward and inverse scattering algorithms based on contrast source integral equations*. Hoboken, NJ: Wiley, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/book/10.1002/9781119741602

[2] Q. Ren, Y. Wang, Y. Li, and S. Qi, *Sophisticated Electromagnetic Forward Scattering Solver via Deep Learning*. Singapore: Springer, 2022. doi: 10.1007/978-981-16-6261-4.

[3] C. Brennan and K. McGuinness, “Site-specific Deep Learning Path Loss Models based on the Method of Moments.” arXiv, Feb. 02, 2023. doi: 10.48550/arXiv.2302.01052.

[4] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*, Second edition. Beijing [China] ; Sebastopol, CA: O’Reilly Media, Inc, 2019.

[5] P. Hennig, M. A. Osborne, and H. Kersting, *Probabilistic numerics: computation as machine learning*. Cambridge New York, NY Melbourne New Delhi Singapore: Cambridge University Press, 2022.