Performance of MATLAB and Python for Computational Electromagnetic Problems

Alec J. Weiss and Atef Z. Elsherbeni

Department of Electrical Engineering
Colorado School of Mines, Golden, Colorado, 80120, United States
aweiss@mines.edu, aelsherb@mines.edu

Abstract **─** MATLAB and Python are two common programming languages commonly used in computational electromagnetics. Both provide simple syntax and debugging tools to make even complicated tasks relatively simple. This paper studies how these programming languages compare in throughput for a variety of tasks when utilizing complex numbers which are common in electromagnetics applications. The compared tasks include basic operations like addition, subtraction, multiplication, and division, along with more complex operations like exponentiation, summation, Fourier transforms, and matrix solving. Each of these tests is performed for both single and double precision on the CPU. A 2D finite difference frequency domain problem and a planar array beamforming problem are also presented for comparison of throughput for realistic simulations.

Index Terms **─** Computational electromagnetics, MATLAB, python.

I. INTRODUCTION

Programming languages such as Python and MATLAB are popular in computational electromagnetics. They provide abstract constructs when compared to lower level compiled programming languages such as C/C++ and FORTRAN. Both Python and MATLAB provide platforms for quickly developing and testing a variety computational electromagnetics (CEM) problems. MATLAB and Python provide optimized libraries that can be leveraged to create computationally efficient programs with a minimal amount of programming. This environment is ideal for the testing of new technologies because users can quickly prototype ideas without worrying about memory management and data types.

A. Current work

Speeds between MATLAB and Python have been previously compared for areas of scientific computing. [1] provided some benchmark tests to compare runtimes on a number of linear algebra routines for real numbers in both Python and MATLAB while articles like [2] and [3] have covered general usage of Python as an alternative to MATLAB for scientific computing. Even further, [4] investigates and discusses the usage of Python for computational electromagnetics (CEM). A vast amount of research has looked specifically on the usage of MATLAB for CEM for general applications like [5] and for more application specific acceleration like in [6].

Unlike MATLAB, Python is a free and open source programming language that is community supported. Python and many scientific computing libraries can be downloaded as a single package like Anaconda [7]. The Anaconda package also includes integrated development environments (IDEs) like Spyder [8] to provide an experience similar to that of working in MATLAB's IDE. For those who are not familiar with Python syntax, there exists many free resources to help learn the intricacies of the programming language and help quickly get a user started. Some resources like [9] even provide direct command translations from MATLAB to Python allowing those familiar with MATLAB to learn how to use numerical libraries in Python even faster.

While MATLAB and Python have both been studied and utilized for a variety of electromagnetic problems, MATLAB still dominates this area of research. *IEEEXplore* has about 3,000 paper matches for the keyword *Python* and over 56,000 for the keyword *MATLAB*. Here we will take an in depth look at how Python matches up as a competitor to MATLAB specifically in CEM problems.

B. Comparison for scientific computing

CEM problems commonly deal in double and single precision complex numbers. To the knowledge of the authors, comparisons of MATLAB and Python for computation using complex numbers has not previously been investigated. Because CEM problems vary so widely in application and implementation, a generic approach to comparison of speeds between MATLAB and Python was taken. For this approach, a large variety of math operations were tested that gradually increase in complexity. These operations are the building blocks for many CEM problems and can be used to estimate the

Submitted On: April 2, 2020
Accepted On: May 21, 2020

relative runtimes between programming languages.

Basic operations (e.g., add, subtract, multiply, divide) are first tested to provide a good baseline for elementary math operations performed on complex numbers. This is then stepped up to include more complex operations like exponentiation, summation, and a combination of elementary operations (specifically $c = a + b * exp(a)$). Past this, matrix operations such as matrix-matrix multiplication and LU decomposition are performed, along with solving of both dense and sparse systems of linear equations. With runtime comparisons for basic operations, relative runtimes can then be estimated for more realistic CEM problems. To test these runtimes, a realistic finite difference frequency domain (FDFD) problem and a beamforming simulation written in both MATLAB and Python were compared.

II. MATLAB AND PYTHON FOR CEM

As previously mentioned, both MATLAB and Python have been used on a variety of applications. Both provide a large range of pre-written optimized functions that the users do not need to rewrite from scratch. In many cases for linear algebra operations, Python and MATLAB can both be configured to use the Intel Math Kernel Library (MKL) [10]. This library provides highly optimized basic linear algebra subprograms (BLAS) and linear algebra package (LAPACK) library for common linear algebra problems. With both programming languages using these subroutines, many linear algebra operations utilize the same precompiled code and therefore would be expected have near identical runtimes.

A. MATLAB

MATLAB arguably provides an easier and more beginner friendly approach for people not experienced in programming. This is because MATLAB does not typically require external libraries. This means that all commands that need to be used in MATLAB are either available from the core installation, or a toolbox. Once a toolbox is installed, the commands from that toolbox are always available to the user. MATLAB also is built specifically for matrix operations. This means that for many CEM applications, the code will be optimized and require less verbose syntax than its Python counterpart.

B. Python

Unlike MATLAB, Python requires libraries to be imported for a variety of tasks. While this provides an extra step by calling the *import* command in Python, it increases the flexibility of the programming language by allowing the user to easily include third party packages without the concern of overlapping function and class names.

While a vast number of Python libraries exist online, three well developed libraries are utilized in this paper. These are *NumPy*, *SciPy*, and *Numba*. These three libraries cover most of the core functionality that MATLAB contains and can therefore be used to solve many CEM problems. *NumPy* and *SciPy* provide functionality such as array and matrix operations, along with access to linear algebra subroutines found in the compiled BLAS backend. *Numba* provides an additional layer of acceleration for Python allowing vector operations to be partially compiled and run around the typical Python interpreter.

C. Further acceleration

Beyond using prebuilt libraries, MATLAB and Python both provide further acceleration capabilities. Both can access functions written and compiled from C/C++ and FORTRAN. In MATLAB, mex file wrappers are written to take MATLAB data types and pass them to these lower level functions. Python on the other hand can typically directly call these compiled functions. This is because at its core, Python typically uses a runtime built in the C programming language. Integration with lower level libraries can therefore be done through a Cython interface, or by loading the functions as a shared library through the ctypes interface.

Python code can also be compiled to native machine code with Cython. This allows users to include additional keywords to specify information such as data types and sizes. This additional information information allows the Cython compiler to optimize the compiled code to provide further acceleration over equivalent native Python code.

While it is important to know these low-level interfaces exist, prototype code will typically be written directly in Python or MATLAB and therefore these interfaces are not quantitatively compared in this paper.

III. MEASUREMENT OF COMPUTATIONAL SPEEDS IN PYTHON AND MATLAB

A. Accurate timing of code

Both MATLAB and Python provide methods to perform some timing analysis on different code snippets with tic()/toc() in MATLAB and timeit() in Python. Extensions of both of these functions were written with new *OperationTimer* classes. The version of this class in MATLAB and Python leveraged the existing timing capabilities but extended upon them to provide more in-depth timing statistics. These classes provided the user with an adjustable number of repeat measurements. In most cases 100 repeat measurements were made.

Repeat measurements are vital as they allow the generation of uncertainty bounds on the data. Bounding the uncertainties of the runtimes helps remove any outliers that may occur due to system inconsistencies (such as operating system scheduling). These uncertainties also provide insight into how variable these runtimes may be over multiple uses. In the case of this paper, these uncertainties were generated as the standard deviation of the repeat measurements.

Along with producing repeat measurements, the *OperationTimer* classes had several other features. This included allowing the user to run a sweep of input data which in turn allowed the testing of runtimes as a function of element count for each operation. Functionality for generating statistics on the data was also included in each class. These classes also provided a consistent interface and data format between both MATLAB and Python to ensure consistency in the post-processing of the results.

B. Testing the functions

With a consistent class for timing of the code in both MATLAB and Python, each of the functions to be compared could then be tested. MATLAB and Python scripts were created that pass the function handles for each operation to test to the *OperationTimer* class. These scripts also set up parameters such as the number of repeat measurements and the sweep dimensions to run for each function. With the scripts set up, each function could then be timed and compared.

IV. CPU RESULTS

CPU results were obtained for the aforementioned operations on an AMD Threadripper 2990WX 32 Core processors with 64 logical processors and 128 GB of RAM. These results were obtained using MATLAB R2018a and Python 3.7 with NumPy 1.17.4, SciPy 1.3.2, and Numba 0.46. Each of the tests was run for both single and double precision complex data. Uncertainty bars were generated using the standard deviation of each of the recorded times.

A. Basic operations

The first set of results were basic add, subtract, multiply, and divide. Each of these operations was repeated 100 times for arrays with a number of elements ranging from 1 all the way to 100 million. The addition operation can be seen in Fig. 1, subtraction in Fig. 2, multiplication in Fig. 3, and division in Fig. 4.

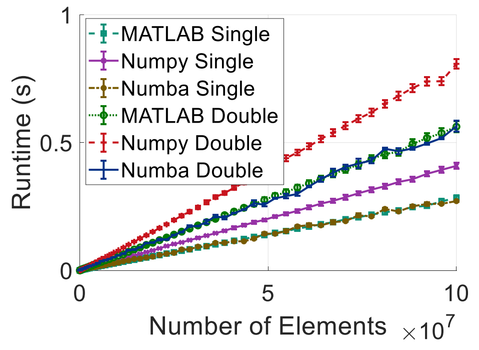


Fig. 1. Runtime vs. number of elements for addition with single and double precision using MATLAB, NumPy, and Numba.

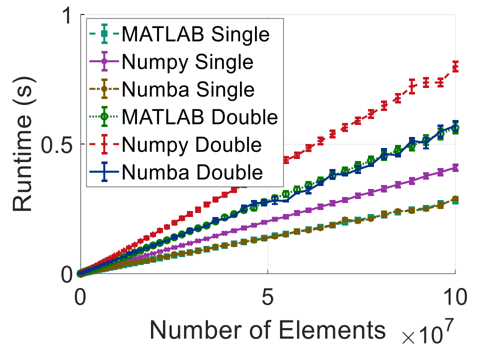


Fig. 2. Runtime vs. number of elements for subtraction with single and double precision using MATLAB, NumPy, and Numba.

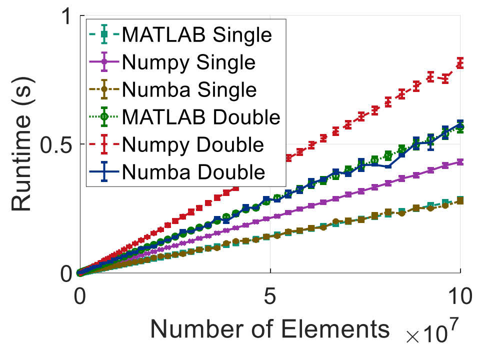


Fig. 3. Runtime vs. number of elements for multiplication with single and double precision using MATLAB, NumPy, and Numba.

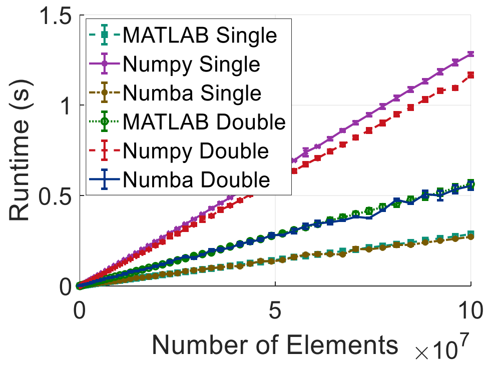


Fig. 4. Runtime vs. number of elements for division with single and double precision using MATLAB, NumPy and Numba.

In each of these cases, both single and double precision for MATLAB and the Python Numba library have almost identical computation time. Numba performs just in time (JIT) compilation of the code to approach speeds of programming languages like C and FORTRAN. MATLAB also reaches these same speeds without any special operations. NumPy on the other hand is much slower with each of these operations. This is most likely because there are number of abstracted steps and calls to the Python interpreter that must be performed on top of the typical computation, causing a slowdown. It is also important to note that both Numba and MATLAB provide a form of parallelization almost no user input (the 'parallel' keyword must be specified for Numba). Depending on the size of the dataset, this also will factor into the speedup of MATLAB and Numba over the Numpy Library.

B. Extended operations

Extended Operations were then tested including exponentiation, a combination of basic operations ($c = a + b * e^{a}$), summation, and the fast fourier transform (FFT). Each of these operations was again run 100 times for a variety of different numbers of elements. All operations except the FFT were run with sizes from 1 to 100 million elements while the FFT was run on arrays of size 1 to 5 million elements. The timed result comparisons of exponentiation and combined operations can be seen in Fig. 5 and Fig. 6 with the sum and FFT seen in Fig. 7 and Fig. 8.

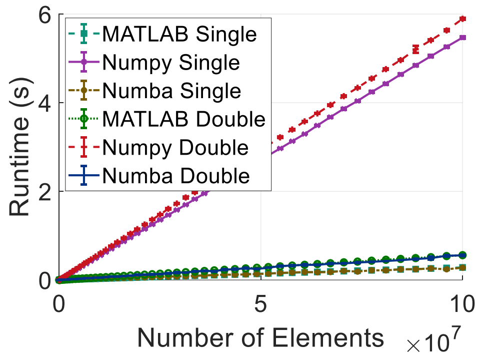


Fig. 5. Runtime vs. number of elements for exponentiation with single and double precision using MATLAB, NumPy, and Numba.

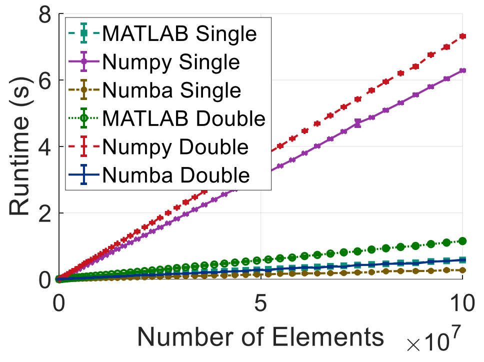


Fig. 6. Runtime vs. number of elements for combined $a + b * exp(a)$ with single and double precision using MATLAB, NumPy, and Numba.

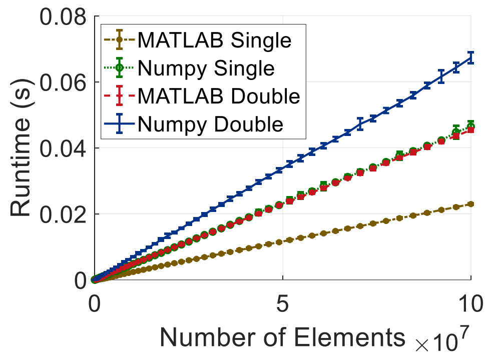


Fig. 7. Runtime vs. number of elements for summation with single and double precision using MATLAB and NumPy.

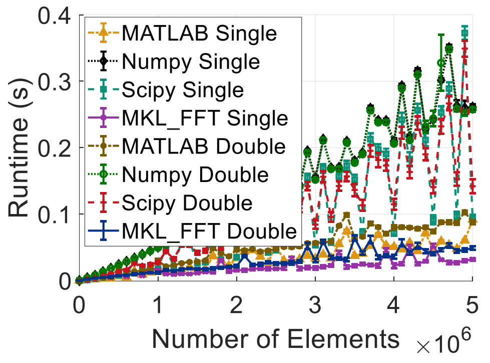


Fig. 8. Runtime vs. number of elements for fft(a) with single and double precision using MATLAB, NumPy, SciPy, and MKL_FFT.

Again, we can see that for exponentiation, MATLAB and Python using Numba provide near identical run times while NumPy proves much slower than the other two. Surprisingly, for the combined operation, Numba outperformed MATLAB for both double and single precision operations. This is most likely because the compiled Numba function does not interact with the Python interpreter, whereas the MATLAB code must return control to its interpreter between each operation. MATLAB again outperforms NumPy in all cases. The FFT operations exemplifies the usage of other libraries in Python, which can be both a strength and weakness. NumPy and SciPy have their own FFT functions while another library called MKL_FFT has yet another implementation. While this increases the difficulty for the user to know exactly what function to use, quick web searches can help find application specific Python libraries that may be faster than those used in MATLAB. It can be seen that while SciPy and NumPy are slower than MATLAB, the MKL_FFT library outperforms MATLAB in terms of speed when performing *fft(a)*.The open source nature also allows the slower libraries to implement parts of the faster ones. In the future, SciPy plans to directly integrate MKL_FFT [11].

C. Matrix operations

The final set of basic operations performed were a set of common linear algebra matrix routines. These routines included matrix-matrix multiplication and LU decomposition along with solving of both dense and sparse linear equations. Matrix multiplication, LU decomposition, and dense matrix solving are all called from the underlying MKL BLAS library and therefore would be expected to perform the same between MATLAB and Python. In Python, the MKL library is used through an interface provided by the NumPy library. The runtime comparisons for matrix multiplication, LU decomposition, and dense matrix solving can be seen in Fig. 9, Fig. 10, and Fig. 11 respectively. As expected, both MATLAB and Python run at almost the exact same speeds for these three operations. In most cases and for most sizes, the runtimes even lie within the uncertainty bounds of one another.

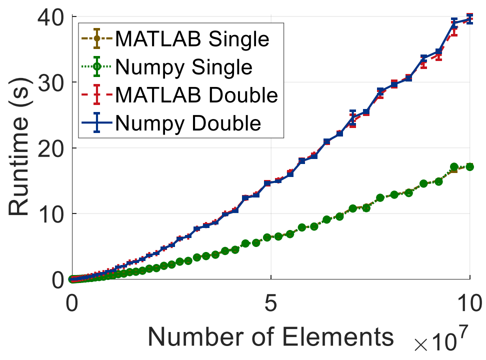


Fig. 9. Runtime vs. number of elements for matrix multiplication with single and double precision using MATLAB and NumPy/SciPy.

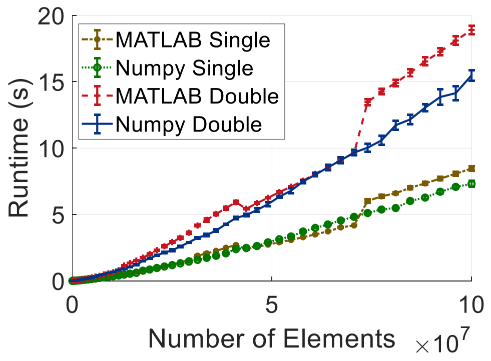


Fig. 10. Runtime vs. number of elements for LU decomposition with single and double precision using MATLAB and NumPy/SciPy.

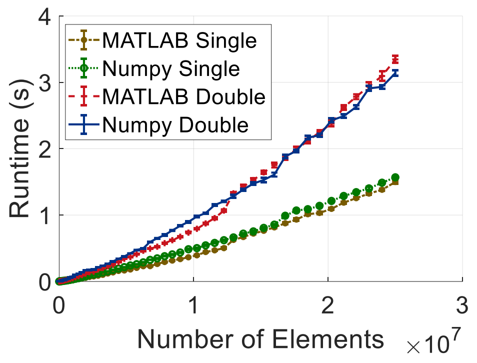


Fig. 11. Runtime vs. number of elements for dense linear equation solving with single and double precision using MATLAB and NumPy/SciPy.

The sparse solving did not fall under the BLAS libraries and therefore there were no expectations for runtime comparisons. MATLAB does not support single precision sparse matrices and therefore only times for double precision for MATLAB are provided. Figure 12 shows the runtimes for a variety of different sizes of sparse matrices. From this plot it can clearly be seen that MATLAB drastically outperforms Python for sparse matrix solving.

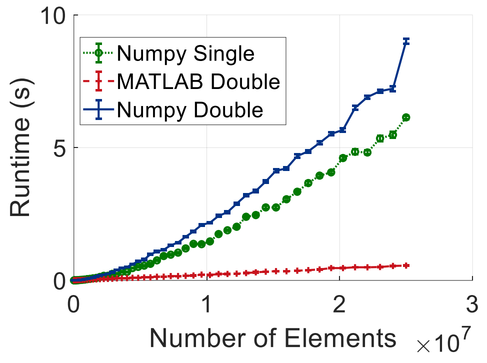


Fig. 12. Runtime vs. number of elements for sparse linear equation solving with single and double precision using MATLAB and NumPy/SciPy.

D. Finite difference frequency domain simulation

While the basic operations tested provide a good basis set of data for runtime comparison between MATLAB and Python, a real FDFD simulation was implemented and the runtimes compared. The FDFD simulation performed calculates the scattering from a 2D cylinder. Because FDFD relies on solving a sparse set of linear equations, it is expected that MATLAB will drastically outperform the runtime of Python because of the runtime comparison of sparse equation solving. The magnitude of the total electric field produced by this simulation can be seen in Fig. 13. These results were produced and verified by both a MATLAB and Python code. The number of cells in the x and y directions were then swept to compare runtimes of the simulations. The runtime as a function of the total number of elements ($cells_x * cells_y$) was then plotted. The runtime comparisons for both single and double precision are given in Fig. 14. Again, MATLAB is not capable of single precision sparse datatypes and therefore this metric is not included.

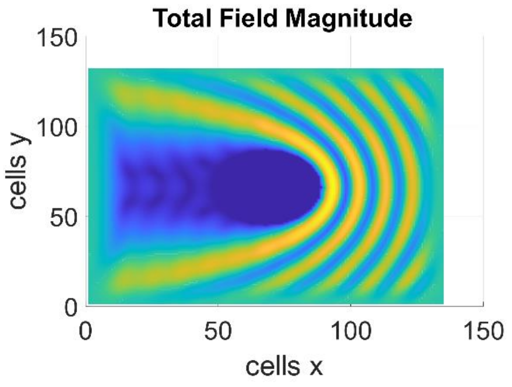


Fig. 13. Total electric field from FDFD scattering from a cylinder problem.

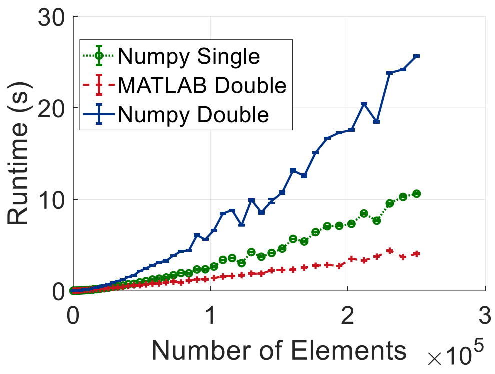


Fig. 14. Runtime vs. number of elements for an FDFD simulation with single and double precision using MATLAB and NumPy/SciPy.

As expected, this data shows that MATLAB drastically outperforms even the single precision version of Python. MATLAB also seems to exhibit a linear increase in runtime ($O(n)$) as the number of cells are increased whereas NumPy exhibits a squared ($O(n^2)$) runtime increase. Because of this it is assumed that for much larger domains, Python FDFD simulations will only become slower compared to MATLAB.

E. Beamforming angle of arrival estimation

A final test was performed with a basic angle of arrival beamforming algorithm. For this test, a large 35 by 35 element antenna array is simulated. An incident plane wave is simulated at a $\pi/4$ radians azimuthal angle and $\pi/4$ radians elevation angle. The output of the beamformed data can be seen in Fig. 15. Beamforming was then performed and times recorded for an increasing number of steering angles. It is expected that because beamforming consists of matrix multiplication and combined elementary operations that Python would outperform MATLAB in this task. The speeds in both MATLAB and Python for a varying number of angles can be seen in Fig. 16.

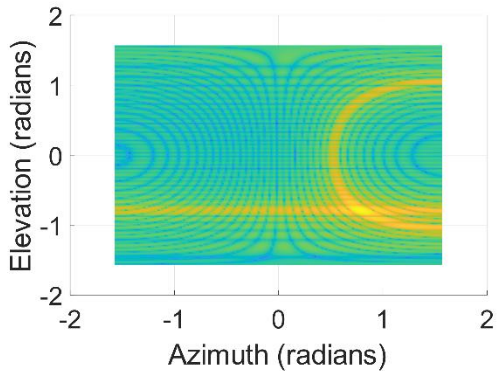


Fig. 15. Beamformed values between -pi/2 and pi/2 radians with 181 calculated angles with an incident plane wave at pi/4 and -pi/4 azimuth and elevation.

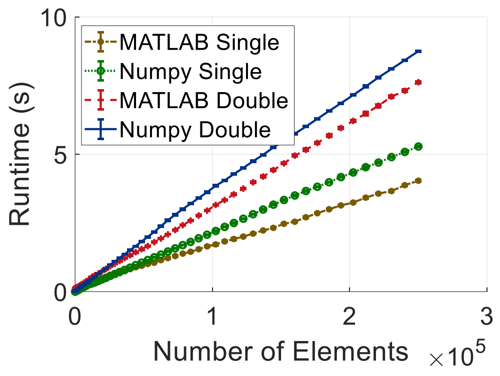


Fig. 16. Runtime vs. number of elements for an antenna array beamforming simulation with single and double precision using MATLAB and NumPy.

Surprisingly, MATLAB still outperformed the Python implementation in this scenario for more than about 20,000 elements. The simulations included setup and array allocation in the timing which may have played a role in the unexpected result. While this is true for large sizes, many times beamforming simulations may be interested in a smaller number of angles and for these cases, Python outperforms MATLAB.

V. CONCLUSION

Although MATLAB is faster for many complex number-based calculations and simulations, other factors come into play when choosing a programming language. For the highest possible speeds in simulations, compiled code written in C or FORTRAN will be the fastest choice. As mentioned earlier, while MATLAB provides accessibility to these lower level functions, Python provides great tools for this integration with ctypes and Cython. Python can also provide acceleration through direct compilation with Cython.

While they were the focus of this paper, runtimes are not the only thing to focus on in a programming language. Python provides a free alternative with many community supported libraries for a variety of different scientific and non-scientific applications. These libraries extend the capabilities of Python to provide everything from easy documentation with sphinx or pydoc, to control of instruments using the pyvisa, pyserial, and socket libraries. While all of these capabilities exist within MATLAB, they typically come as an additional cost in a MATLAB toolbox.

While all of these factors should be taken into consideration before deciding on a programming language, MATLAB clearly outperforms Python in the majority of complex math operations and all CEM simulations tested. That being said, Python can provide a well-supported alternative to MATLAB for usage in CEM problems.

REFERENCES

[1] J. Unpingco, “Some comparative benchmarks for linear algebra computations in Matlab and Scientific Python,” in *2008 DoD HPCMP Users Group Conference*, pp. 503-505, 2008. doi: 10.1109/DoD.HPCMP.UGC.2008.49.

[2] R. Python, “MATLAB vs Python: Why and how to make the switch – Real Python,” [Online]. Available: https://realpython.com/matlab-vs-python/. [Accessed: 30-Dec-2019].

[3] J. Ranjani, A. Sheela, and K. P. Meena, “Combination of NumPy, SciPy and Matplotlib/Pylab - A good alternative methodology to MATLAB - A comparative analysis,” in *2019 1st International Conference on Innovations in Information and Communication Technology (ICIICT)*, pp. 1-5, 2019 doi: 10.1109/ICIICT1.2019.8741475.

[4] N. Kinayman, “Python for microwave and RF engineers [Application Notes],” *IEEE Microwave Magazine*, vol. 12, no. 7, pp. 113-122, Dec. 2011. doi: 10.1109/MMM.2011.942704.

[5] M. Capek, P. Hazdra, J. Eichler, P. Hamouz, and M. Mazanek, “Acceleration techniques in Matlab for EM community,” in *2013 7th European Conference on Antennas and Propagation (EuCAP)*, pp. 2639-2642, 2013.

[6] A. J. Weiss, A. Z. Elsherbeni, V. Demir, and M. F. Hadi, “Using MATLAB’s parallel processing toolbox for multi-CPU and multi-GPU accelerated FDTD simulations,” vol. 34, no. 5, p. 7, 2019.

[7] “Anaconda | The world’s most popular data science platform,” *Anaconda*. [Online]. Available: https://www.anaconda.com/. [Accessed: 14-Jan-2020].

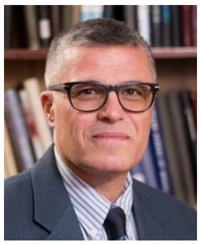
[8] “Spyder Website.” [Online]. Available: https://www.spyder-ide.org/. [Accessed: 14-Jan-2020].

[9] V. Gundersen, “NumPy for MATLAB users.” 2006.

[10] ajolleyx, “Intel® Math Kernel Library (Intel® MKL),” 00:00:14 UTC. [Online]. Available: https://software.intel.com/en-us/mkl. [Accessed: 13-Jan-2020].

[11] “SciPy Roadmap — SciPy v1.4.1 Reference Guide.” [Online]. Available: https://docs.scipy.org/doc/scipy/reference/roadmap.html. [Accessed: 16-Jan-2020].

Alec J. Weiss received his B.S. degree in Electrical and Computer Engineering from the University of Colorado, Boulder, Colorado, USA in 2017 and his M.S. in Electrical Engineering from the Colorado School of Mines, Golden, Colorado, USA in 2018 where he is currently pursuing his Ph.D. in Electrical Engineering. He joined the National Institute of Standards and Technology (NIST) Communications Technology Laboratory (CTL) in 2017 as a graduate student researcher. His research interests include millimeter-wave measurements, 5G communications systems, and high performance computing for electromagnetic applications.



Atef Z. Elsherbeni received an honor B.Sc. degree in Electronics and Communications, an honor B.Sc. degree in Applied Physics, and an M.Eng. degree in Electrical Engineering, all from Cairo University, Cairo, Egypt, in 1976, 1979, and 1982, respectively, and a Ph.D. degree in Electrical Engineering from Manitoba University, Winnipeg, Manitoba, Canada, in 1987. He started his engineering career as a part time Software and System Design Engineer from March 1980 to December 1982 at the Automated Data System Center, Cairo, Egypt. From January to August 1987, he was a Post-Doctoral Fellow at Manitoba University. Elsherbeni joined the faculty at the University of Mississippi in

August 1987 as an Assistant Professor of Electrical Engineering. He advanced to the rank of Associate Professor in July 1991, and to the rank of Professor in July 1997. He was the Associate Dean of the College of Engineering for Research and Graduate Programs from July 2009 to July 2013 at the University of Mississippi. He then joined the Electrical Engineering and Computer Science (EECS) Department at Colorado School of Mines in August 2013 as the Dobelman Distinguished Chair Professor. He was appointed the Interim Department Head for (EECS) from 2015 to 2016 and from 2016 to 2018 he was the Electrical Engineering Department Head. In 2009 he was selected as Finland Distinguished Professor by the Academy of Finland and TEKES. Elsherbeni is a Fellow member of IEEE and ACES. He is the Editor-in-Chief for ACES Journal, and a past Associate Editor to the Radio Science Journal. He was the Chair of the Engineering and Physics Division of the Mississippi Academy of Science, the Chair of the Educational Activity Committee for IEEE Region 3 Section, the General Chair for the 2014 APS-URSI Symposium, the president of ACES Society from 2013 to 2015, and the IEEE Antennas and Propagation Society (APS) Distinguished Lecturer for 2020-2022.