12-1-2018

Developing a Python-based Workflow to Modernize User Interaction with Legacy Programs

Robert H. Hunter
Mississippi State University

Developing a Python-based Workflow to Modernize User Interaction with Legacy Programs

By

Robert H. Hunter

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Bachelors of Science
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

December 2018

Developing a Python-based Workflow to Modernize

User Interaction with Legacy Programs

By

Robert H. Hunter

Approved:

Matthew W. Priddy
(Major Professor)

Seth Oppenheimer
(Committee Member)
Honors College Representative

Mahalingam Ramkumar
(Committee Member)

Name: Robert H. Hunter

Date of Degree: December 14, 2018

Institution: Mississippi State University

Major Field: Computer Engineering

Major Professor: Dr. Matthew W. Priddy

Title of Study: Developing a Python-based Workflow to Modernize User Interaction with Legacy Programs

Pages of Study: 34

Candidate for Degree of Bachelors of Science

This thesis explores the process of refactoring legacy scientific software to improve overall usability and the ease with which a researcher can modify the software. The legacy code is Fortran code used to run different mechanical loading scenarios on a user-defined material subroutine. Each material is represented by a Fortran subroutine, and a separate Fortran program was used to call the material. To upgrade this program without sacrificing too much performance and functionality, the material subroutine is compiled into a Python library using f2py, and the driver code is translated into Python. This paper covers this upgrade process, the impact it had on results, and how it improved analysis of the output data.

Key words: Fortran, Python, f2py, Legacy Code, Abaqus, Jupyter Notebook

DEDICATION

To my parents, Bobby and Lindy Hunter.

ACKNOWLEDGEMENTS

I want to thank the Shackhouls Honors College for providing the resources and support that enhanced my academic career here at Mississippi State University and made this research project possible. The advising, academic rigor, and financial support the College provided has been invaluable to my experience as an undergraduate student.

I thank Dr. Seth Oppenheimer for his support as a mentor and representative of the Provost Scholar program.

Last, but not least, I would like to thank Dr. Matthew Priddy for giving me the opportunity to do undergraduate research and for directing the completion of my thesis project.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LISTINGS v

LIST OF FIGURES vii

CHAPTER

1. INTRODUCTION 1

    1.1 Motivation 1
    1.2 Approach 2

2. EXPERIMENT 4

    2.1 Problem Background 4
    2.2 Solution 6
    2.3 Rewriting Fortran 7
        2.3.1 Explicit Control Transfer 7
        2.3.2 Indexing 11
    2.4 Fortran to Python Interface Generator 12
        2.4.1 F2PY 12
        2.4.2 Basic Use 13
        2.4.3 Application to the Experiment 14

3. VISUALIZATION 16

    3.1 Timing 16
    3.2 Interactive Notebooks 17
    3.3 Generating Data 18
    3.4 Plotting 19
    3.5 Cloud-based solution 22

4. CONCLUSION 25

4.1 Advantages 25
4.1.1 Platform-Independence 25
4.1.2 Increased Functionality 26
4.1.3 Adaptability 26
4.1.4 Collaboration 27
4.2 Future Work 27

REFERENCES 29

APPENDIX

A. CODE SNIPPETS 32

LISTINGS

2.1 Python - MatPointSim.py 10

2.2 Fortran - MatPointSim.f 10

2.3 MatPointSim.f 11

2.4 MatPointSim.py 11

A.1 Iterating UMAT Call 33

A.2 Plotting Stress 34

A.3 Deformation Plot 35

LIST OF FIGURES

2.1 Flowchart of MatPointSim.f 5

2.2 Process used to refactor MatPointSim.f 8

3.1 Test Case Performance of Python and Fortran 17

3.2 Stress Tensor 20

3.3 Strain Tensor 20

3.4 Stress at different loading directions 21

3.5 Deformation resulting from MatPointSim.py 23

CHAPTER 1

INTRODUCTION

1.1 Motivation

The field of computer programming has now been around for over 65 years [1]. Many different methods and machines have been involved in the practice of solving problems using electronic computers. Early in the development of computers, programming languages were created to simplify the task of writing instructions for computers to execute. Since their inception, programming languages have evolved from low-level assembly code into hundreds of different languages with a wide range of syntax and methods of compilation [2]. Despite the immense advancements in programming that has been made over the past two decades, many universities, research institutions, and engineering firms still run programs written in languages that are over 40 years old.

While these programs can still be very effective at accomplishing their intended task, there are several downsides to continuing to use these programs in their original language. For one, legacy languages like Fortran 77 can be quite cryptic to the average entry-level researcher today. While many researchers would resist learning to operate a tool written in an unfamiliar language like Fortran, they would be much more willing to work with programs written in a language with widespread use today.

Another issue with legacy code is adaptability. A tool may be able to compute perfect results for every iteration, but the old method of printing these results to a shell's `stdout` or saving them to a custom-formatted plain text file makes it far from convenient to analyze results. Modern languages supply more string manipulation methods that make it easier for the average researcher to modify the program to present the results in the most suitable fashion. Languages like Python and R even provide native packages to visualize results without having to pipeline the data to another tool.

How can researchers continue to keep legacy code bases relevant when there are so many advantages to using modern scripting languages? This is the key question that motivated this research project. The aim of this research is to examine the solutions that currently exist and explore how an ideal solution is applied to a specific project.

1.2 Approach

The process for replacing or refactoring legacy code in research institutions should be convenient and efficient to allow researchers to take advantage of it. Given the modern push toward cloud-computing [3] [4], a solution could be a cloud-based tool that allowed researchers to share the updated code and collaborate over the analysis. This tool should use a modern language that has been widely adopted in the scientific computing community. Python has been used for scientific computing since 1995 [5]. While the choice of a programming language ultimately depends on the needs of the project, Python comes with a wealth of libraries and packages that make it useful for a wide range of projects. Ideally, a researcher could upload a Fortran program to a server, convert the code into a Python

module, run the code on the server, and share the results with collaborators through this cloud-based environment. This was the approach taken for this research project. Using `f2py` and Jupyter notebooks, a web-based workflow for converting, running, and analyzing a legacy code base was developed.

CHAPTER 2

EXPERIMENT

2.1 Problem Background

Abaqus is a finite element (FE) software package commonly used to solve structural, thermal, and multi-physics analyses. One of the unique features of Abaqus is that it allows the user to specify information (e.g. stress/strain response) about certain material features through a subroutine. It does this by providing an interface for Fortran user-defined material subroutines (or UMATs) [6]. These UMATs determine the updated tangent stiffness matrix and multiply it by the strain increment to determine the updated stress. One major issue with this workflow is the inability to easily debug these UMAT files before generating results with them in Abaqus. While Abaqus is great at providing the final simulation results, it does not display any intermediary information that would be helpful in testing the accuracy of the UMAT subroutines. Fully debugging the UMATs would require iterating over select input parameters to understand the complete material response. To supply some level of debugging, the Mechanical Engineering Department at MSU uses another Fortran file, `MatPointSim.f`, that is meant to replace the function of Abaqus in this scenario. It runs loading scenarios (e.g. tension, compression, and torsion) through the UMAT procedure and displays the UMAT's direct output (Figure 2.1).

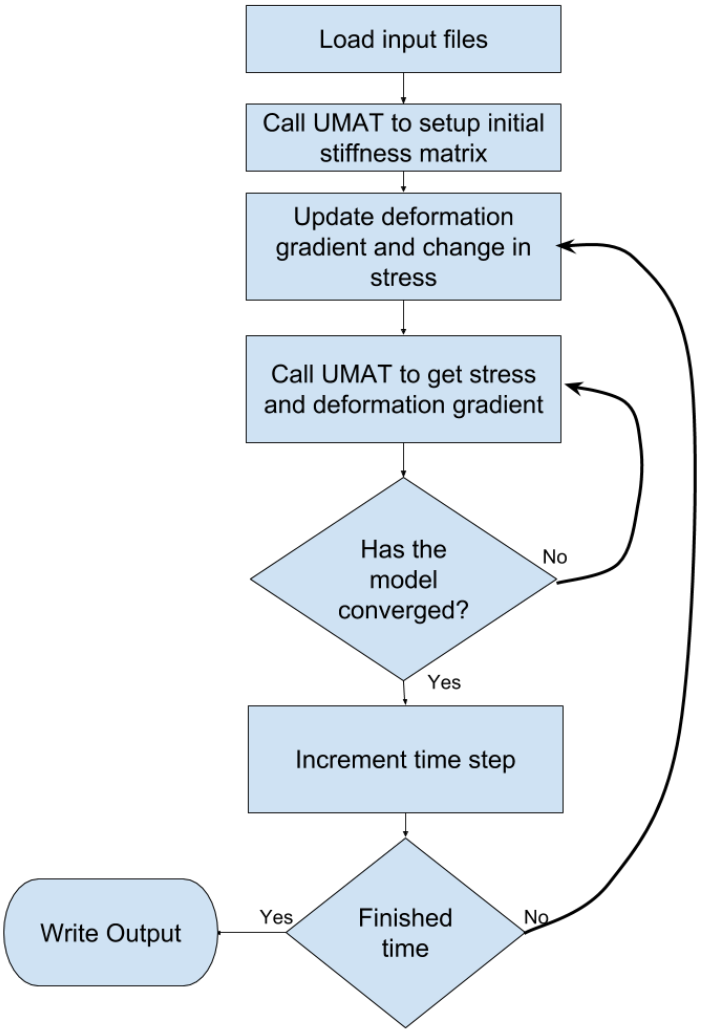

Figure 2.1

Flowchart of MatPointSim.f

This method makes it much easier to test the validity of the UMAT before calling it from Abaqus. However, because UMAT files are written in Fortran, `MatPointSim.f` is also written in Fortran, which means this approach to running the UMATs suffers from all the disadvantages of legacy code programs described in the introduction. `MatPointSim.f` specifically is compiler-dependent, uses an undocumented text file as input, and prints results to an unformatted text file as well as `stdout`. To visualize or analyze the result data, the output from the program would have to be exported and loaded into a separate tool or code base. Adding these extra steps to a testing workflow is far from optimal. In short, there needs to be a method to run/debug these UMATs in a cross-platform development environment, analyze the results in a language accessible to the average technical researcher, and create helpful visualizations of these results with ease.

2.2 Solution

The kind of problem just described is encountered everywhere in scientific research today [7]. Legacy code is a common occurrence now that the sphere of computational engineering has been around for well over half a century. Researchers are reluctant to directly edit code bases that have been "tried-and-true" sources of accurate results for decades. Not only can it require a steep learning curve to translate these legacy code bases into modern code, but a lot of times it inhibits performance. Modern languages are written more generally than older languages, and are more dependent on the vast computing power of modern machines. Older compiled languages are written at a very low-level of abstraction and were meant for computers with very little resources. Thus, compiled programs from Fortran or

C will be much faster than similar scripts written in Python or Ruby [8]. Because of this, an ideal method for refactoring legacy code would involve preserving the performance and structure of the original code while providing an interface for modern scripting languages. This would resemble some kind of wrapper-based solution. A wrapper handles all of the program's I/O and calls the original subroutine in order to do the actual computation.

To apply this approach to running the UMATs, `MatPointSim.f` had to be rewritten into Python to call each UMAT Fortran subroutine through a Python wrapper. This approach not only allowed the wrapper method to be explored and tested, but also provided a way to compare this method with directly rewriting Fortran into Python, since `MatPointSim.f` had to be translated into Python code. Figure 2.2 shows how both of these methods were applied to develop a Python version of the UMAT debugging process.

2.3 Rewriting Fortran

To run the Python generated wrappers, `MatPointSim.f` had to be converted from Fortran code into a Python script. While Fortran and Python are syntactically similar enough to make this a fairly straightforward process, there are some foundational differences between the languages that provided some obstacles. This experience demonstrated further the convenience of using a tool like `f2py` instead of trying to convert Fortran to Python every time.

2.3.1 Explicit Control Transfer

One programming mechanism that was once a common concept in high-level languages is the idea of *explicit control transfer* [9]. This mechanism allows the programmer

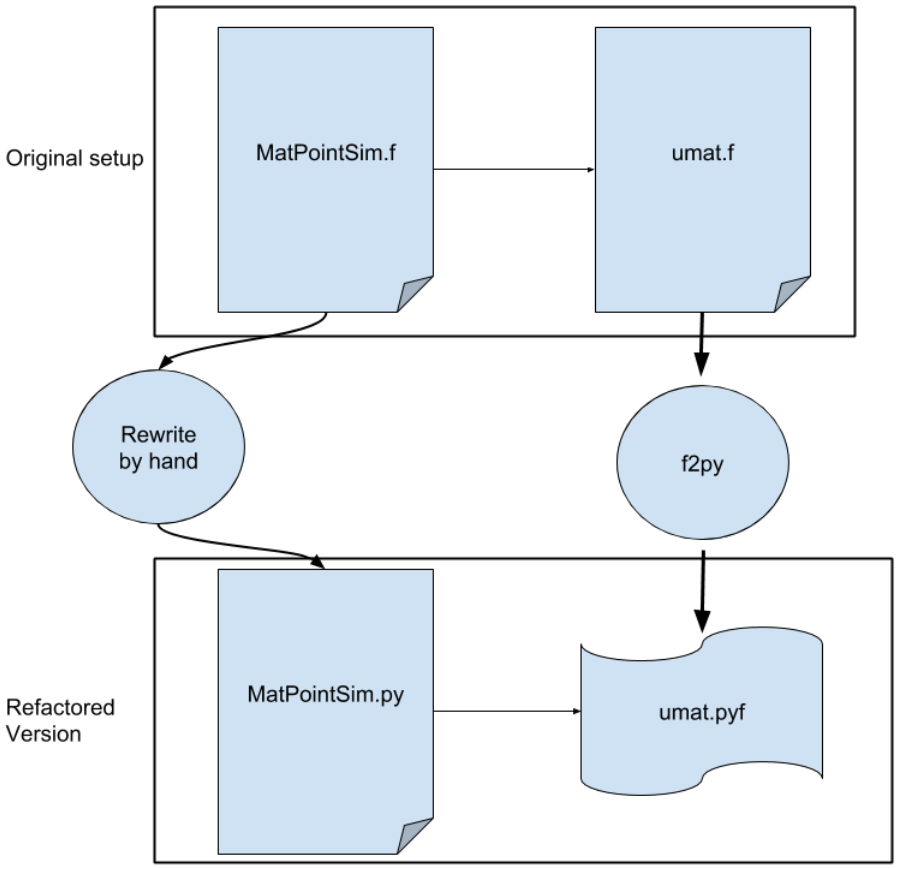

Figure 2.2

Process used to refactor MatPointSim.f

to directly command the program to jump from one line to another. In Fortran this is manifested in `goto` statements, which allow you to jump to a specified `label`. While `goto` (or `jump`) statements were once a regular part of programming, their use was eventually discouraged [10] [9]. Control structures such as loops, breaks, and if/else statements are sufficient to satisfy the role the `goto` statement once fulfilled [11]. However, rewriting a program to use these control structures instead of `goto` statements is not always straightforward. Sometimes the labels and `goto` statements are nested in ways that don't translate directly into loop or branching structures. Often, these statements will have to be shifted or rearranged before they can be converted into loops and if/else statements [11]. In this experiment, two labels and goto statements were used in `MatPointSim.f`. One set was to iterate over the UMAT until a stress values converged to within a set tolerance, and the second label set repeated the entire process over a specified number of time steps. Multiple `goto` statements pointed to each of these labels, so the control sequence did not consist in a single conditional loop or branch. While it often seems impossible to replace these unstructured control transfers with loops and if statements, in reality there is an algorithmic process described by Erosa and Hendren that can be used to eliminate `goto` statements [11]. A similar process was followed to replace the `goto` statements in `MatPointSim.f`. Here is a snippet of the code changes that were used to eliminate `goto` statements from the Fortran code:

Listing 2.1

Python - MatPointSim.py

```
while True: # Label 400
    time[0] = time[0] + dtime
    ...
    while True: # Label 500
        if Kinc > MAX_ITR:
            time[0] -= dtime
            time[1] -= dtime
            dtime = dtime / 2.0
            cont = True
            break
        Kinc += 1
        ...
    if cont: continue
```

Listing 2.2

Fortran - MatPointSim.f

```
400 time(1) = time(1) + dtime
    ...
    Kinc = 0
500 if (Kinc .ge. Max_itr) then
      time(1) = time(1) - dtime
      time(2) = time(2) - dtime
      dtime = dtime / 2
      go to 400
    end if
    Kinc = Kinc + 1
    ...
```

As the code demonstrates, each label was replaced with a `while` loop header. The difficult part came at eliminating the line

```
go to 400
```

in the Fortran code, because this `goto` is in a nested loop (label `500`). To allow this `goto` to completely exit the inner loop and restart the outer loop, the boolean flag `cont` was added. If this flag was set when the loop completed (the `break` was hit), then a `continue` statement was used to restart the outer loop.

While the process of eliminating `goto` statements requires some care (not to mention a fair amount of debugging), it is indeed doable and intuitive. It is something that will be required often when translating legacy code into modern high-level languages.

2.3.2 Indexing

Another complexity that was encountered in converting `MatPointSim.f` into `MatPointSim.py` was replacing Fortran's one-based array indexing with Python's zero-based array indexing. Most of situations where indexing was an issue, it was simply a matter of subtracting one from an index value whenever an array was accessed in the Python. However, some transformations were needed when the Fortran used certain operations on index values. An example of this was when `MatPointSim.f` used the modulo operator to modify the loading directions based on the mechanical operation. To have `k` be a valid Python index but still result in the expected use of the modulo operator, the math had to be altered slightly:

Listing 2.3

MatPointSim.f

```
if (lm .eq. 'TC') then
    ...
    k = j
    i = mod(k   ,3)+1
    j = mod(k+1,3)+1
```

Listing 2.4

MatPointSim.py

```
if loading_mode == "TC":
    ...
    k = loading_j - 1
    i = loading_j % 3
    j = (loading_j+1) % 3
```

The index of the first element is not the only thing that is different about Fortran's array indexing. For multi-dimensional arrays, Fortran uses "column-major" ordering instead of "row-major" ordering [12]. This means that when a 2D array (a matrix) is accessed, the column index is the first one specified.

- Column-major order: `array(col, row)`
- Row-major order: `array[row][col]`

Row-major ordering is used in most common programming languages like C or Python's numpy library [13]. Thus, using a consistent index order is necessary when writing Python code that is supposed to call Fortran routines. Python's numpy library will raise a warning whenever a Fortran subroutine is called with a numpy array that uses row-major order instead of column-major order. To remedy this, the Python program (in the case of this project, `MatPointSim.py`) has to merely convert all numpy arrays into Fortran-style numpy arrays. The numpy method `numpy.asfortranarray` accomplishes this exact task [14]. The code in `MatPointSim.py` runs any numpy arrays through this method to ensure that the arrays use the same index ordering that Fortran will expect.

2.4 Fortran to Python Interface Generator

Once `MatPointSim.f` was rewritten as `MatPointSim.py`, the only step that remained was to determine how to call the Fortran UMAT subroutines from a Python script. While there are a few different software tools available for interfacing different languages with Python, the tool selected for this project is a library aptly named, f2py.

2.4.1 F2PY

Pearu Peterson first released `f2py` as a single Python script in 1999 as the "Fortran to Python Interface Generator" (FPIG) [15]. It became part of Python's `numpy` bundle in 2007, when it began to see more widespread use. The goal of `f2py` is to create a software tool that

- No prior knowledge of mixed-programming techniques should be required to create and to use the interfaces between Python and Fortran programs.

- The development of a Fortran library of subprograms and a Python program using that library, should be independent. That is, neither the Fortran nor the Python user need not to be familiar with the other language.
- Creating robust and immediately usable wrappers between Fortran and Python programs should be automatic and triggered by a single command. However, there should also be an opportunity to tune the interfaces.
- The interface generator tool should take advantage of the information available in the Fortran source code and create as Pythonic an interface as possible.
- The interfacing solution should be as easy to use for large Fortran libraries (with thousands of subprograms) as for a simple Fortran procedure (Fortran function or subroutine). [16] [17]

The primary function of `f2py` is compiling Fortran files into a Python library file that can be imported into another Python program. The `f2py` compiler completes this process by parsing the Fortran code to determine argument types and intents (e.g. input variables and output variables), compiling the Fortran code with the environment's standard Fortran compiler, and generating Pythonic wrappers for the compiled targets that account for the argument types determined during the parse stage [17].

2.4.2 Basic Use

The complexity of using f2py depends on the project to which it is being applied. Some basic Fortran subroutines can be compiled directly without the f2py user having to make any intermediate specifications. However, in the majority of cases, the tool will require the user to specify at least the `intent` of arguments to each subroutine for it to compile correctly [17]. While recent versions of Fortran have the programmer specify the intent of subroutine arguments in the code, programs written in past versions will require the f2py

user to specify the intents in a special file called the "signature file". These intents can be one or more of the following options [17]:

- `in`: Input parameters. These parameters will be treated as input parameters that are not changed in the subroutine.
- `out`: Output parameters. All parameters labeled with this intent will be returned in a list at the end of the subroutine.
- `inplace`: Parameters labeled "inplace" are input/output variables whose value will be modified in place. This is similar to pass-by-reference in C++.
- `copy`: A variation of the `in` class. These parameters are treated as inputs, but copies of them will be made for the subroutine so the original value isn't altered.

There are a few other intent options that can be used as well; however, the above covered the needs of this project.

2.4.3 Application to the Experiment

F2py was used to compile the UMAT Fortran subroutines into Python bindings for `MatPointSim.py`. This process required a few different steps to complete. First, all of the inline comments (comments starting with `!`) had to be removed before it would successfully compile the program due to the version of Fortran f2py used. However, this was the only time the syntax of the UMAT had to be modified. Theoretically, if the versions of Fortran were the same, the Fortran code would not have to be edited. Once the file was compiling successfully, the only further step was to assign intent attributes for the main variables of the subroutine. F2py was used to generate a signature file for the subroutine, and this signature file was edited to specify the input/output intent of the UMAT arguments. While most of the parameters were untouched (leaving them as `intent(in)` arguments)

the primary output variables were specified as `intent(out)`. As described earlier, this meant that they were all returned from the function in a tuple.

Once the signature file was setup correctly (this required very minimal editing), the original UMAT file was compiled along with the signature file. `f2py` generated the Python.

CHAPTER 3

VISUALIZATION

3.1 Timing

One of the disadvantages of converting a program from compiled code to a scripted language is the performance. Python executes by compiling a Python script into virtual machine code that it then executes in line-by-line fashion [18]. This virtual machine code must be executed by the Python interpreter, which gives it the advantage of being platform-independent. Any operating system that has a Python interpreter installed can execute Python virtual machine code [19]. However, because this virtual machine code cannot be run directly on the operating system like assembly language, it has a much higher performance overhead than compiled languages like C or Fortran. This difference becomes evident when comparing `MatPointSim.f` to `MatPointSim.py`. Although the UMAT code is in compiled Fortran for both programs, `MatPointSim.py` takes on the overhead of calling the UMAT from Python virtual machine code. To measure the exact difference, both programs were timed and recorded for five runs using a simple tension and compression loading scenario. Figure 3.1 shows the performance advantage of the pure Fortran.

While exact performance difference is obviously machine and operating system dependent, these timing measurements show a 300% increase in time to run the Python version

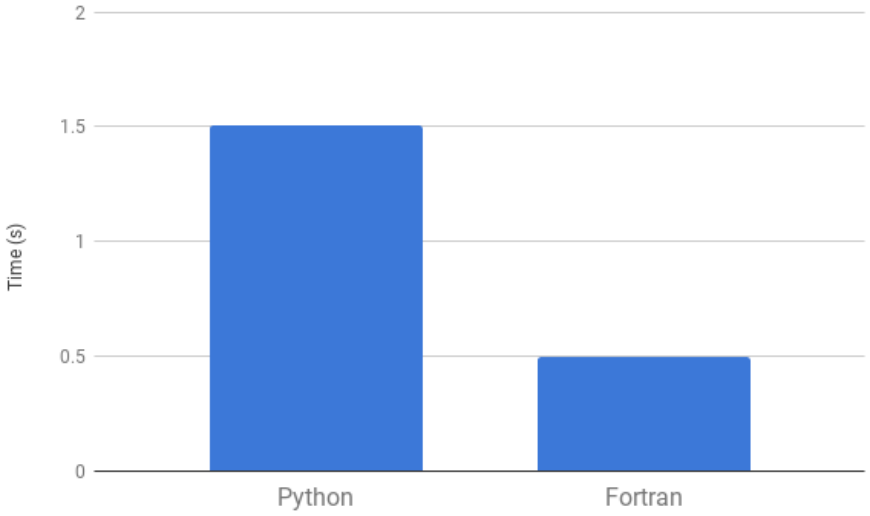

Figure 3.1

Test Case Performance of Python and Fortran

of the code. While this only results in a single second of extra time in the simple use case of the UMAT program, it demonstrates how the overhead of a modern scripting language can negatively impact performance.

3.2 Interactive Notebooks

One of the major advantages of using a modern scripting language like Python for scientific computing is the wide variety of software tools and libraries that are readily available. Jupyter (previously iPython) is one such tool [20]. Jupyter is a web-based development environment that allows users to write snippets of code and see them run interactively in a web browser [20]. A researcher can create a Jupyter notebook and start analyzing data in the browser, getting interactive results in the form of plots, tables, and even other html widgets. The user can even execute certain snippets of code separately from the whole,

so certain results can be refreshed without having to run the entire notebook again. This workflow is ideal for all kinds of simulations, particularly those that involve a lot of processing and data generation [21]. Using Jupyter notebooks is an easy way to demonstrate the advantages using f2py to convert Fortran simulations into a language like Python.

Once `MatPointSim` was converted to a Python program that called f2py-compiled subroutines, Jupyter notebooks were used to run the program and interact with the results. All of the visualizations and experiments demonstrated in the next few sections were generated with the use of Jupyter notebooks.

3.3 Generating Data

Converting `MatPointSim` into Python made it much easier to modify the code to generate new sets of data. By importing the main function of `MatPointSim.py` into a Jupyter notebook, it became possible to iterate over several new input variables to conduct more thorough tests than were run before. The first variable to be tested was the loading direction. Listing A.1 in the appendix contains the code used to iterate over all nine loading directions, run `MatPointSim` for each loading scenario, and save the stress and strain from each run. From this code it only takes a few lines of code to then plot the stress tensors for each of these loading scenarios, as demonstrated in Listing A.2. This results in the plots shown in Figure 3.4. The component that receives the most stress can be seen to vary depending on the direction the load is applied. In this and all of the following plots, the stress and strain tensors are arrays that represent values in the following stress/strain matrix:

$$\sigma = \begin{bmatrix} S_1 & S_4 & S_5 \\ \dots & S_2 & S_6 \\ \dots & \dots & S_3 \end{bmatrix} \tag{3.1}$$

These plots allow the user to run a UMAT and see whether it is providing the expected results. Several different variables can be iterated over, and the material response can be quickly and compactly plotted to determine the UMAT's accuracy. This is a vast improvement over digging through tables of text output in a file.

3.4 Plotting

Once the code to call `MatPointSim.py` and generate data has been written, the notebook allows the user to experiment with different visualization techniques without having to rerun `MatPointSim`'s main subroutine over and over again. Because of the flexibility of Jupyter notebooks, rendering new plots is simply a matter of changing the input file and rerunning the notebook. The loading scenario in the included plots involves simultaneous compression and torsion. Figure 3.2 through Figure 3.4 demonstrate the kind of plots that can be generated from `MatPointSim.py` in a Jupyter notebook. Loading directions cannot be the same in the torsion and compression scenario, which is why plots in the diagonal of Figure 3.4 are missing.

Python and Jupyter have tools that can be used to plot even 3D meshes. Using the `plotly`[22] Python library, plots containing 3D elements can be rendered inside of a Jupyter notebook. This can be used to visualize the deformation gradient generated by

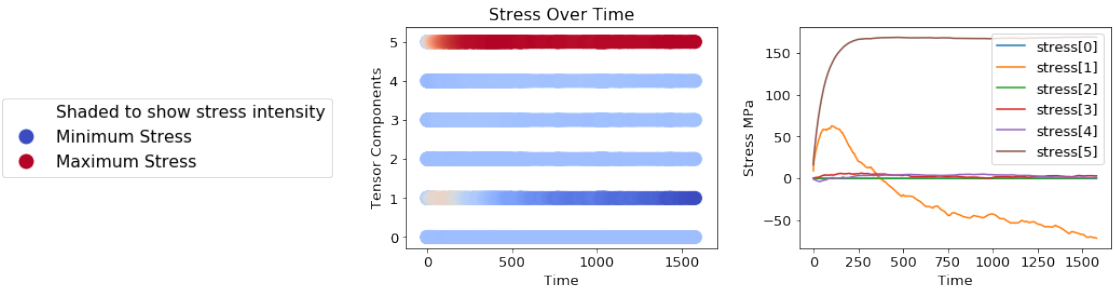

Figure 3.2

Stress Tensor

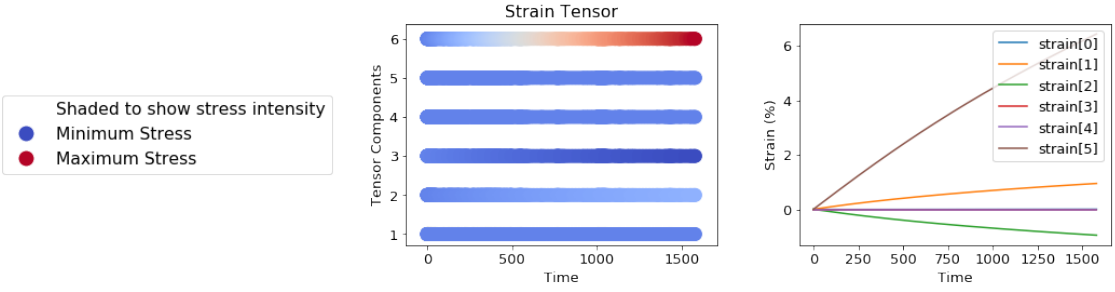

Figure 3.3

Strain Tensor

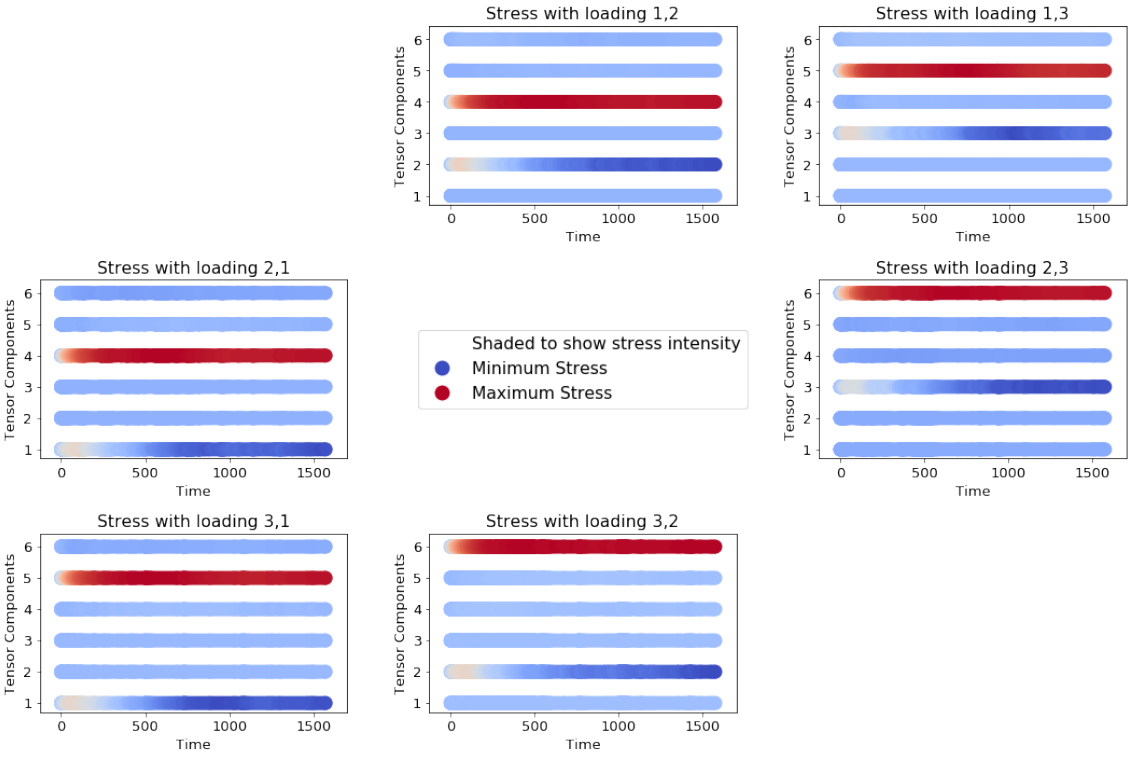

Figure 3.4

Stress at different loading directions

the UMAT. This is done by applying the deformation gradient matrix to the vertices of a cube rendered in plotly [23] [24]. Figure 3.5 demonstrates the deformation loading that `MatPointSim.py` performs in a simultaneous torsion and compression loading scenario. Most of the code for this plot involves mapping the deformation gradient generated by the UMAT to cube vertices. Listing A.3 demonstrates the conciseness of the plotting routine using plotly in a Jupyter notebook.

3.5 Cloud-based solution

Jupyter notebooks run from a web server normally hosted on the client's machine. However, because Jupyter is a web-based tool, it is perfectly possible to launch a Jupyter server on a remote machine and therefore permit multiple people to access the same Jupyter notebook from two different machines. A researcher can use this feature to give a colleague access to a hands-on analysis of research results. Not only can a researcher see the plots and figures that a colleague is generating, he/she can modify the program interactively to change the presentation of results or conduct different analysis in the notebook.

This aspect of Jupyter notebooks can be applied to the UMAT project to make it a cloud-based process. Using a Jupyter server on a remote machine, a researcher can upload their UMAT to the server and run a specified notebook to convert the UMAT and automatically generate plots to test it. Jupyter notebook allows operating system commands to be run by preceding the line with a "`!`". Thus, the Python module for a UMAT can be generated in a notebook with the single line:

```
!f2py -c plasticity_t.pyf plasticity_umat.f
```

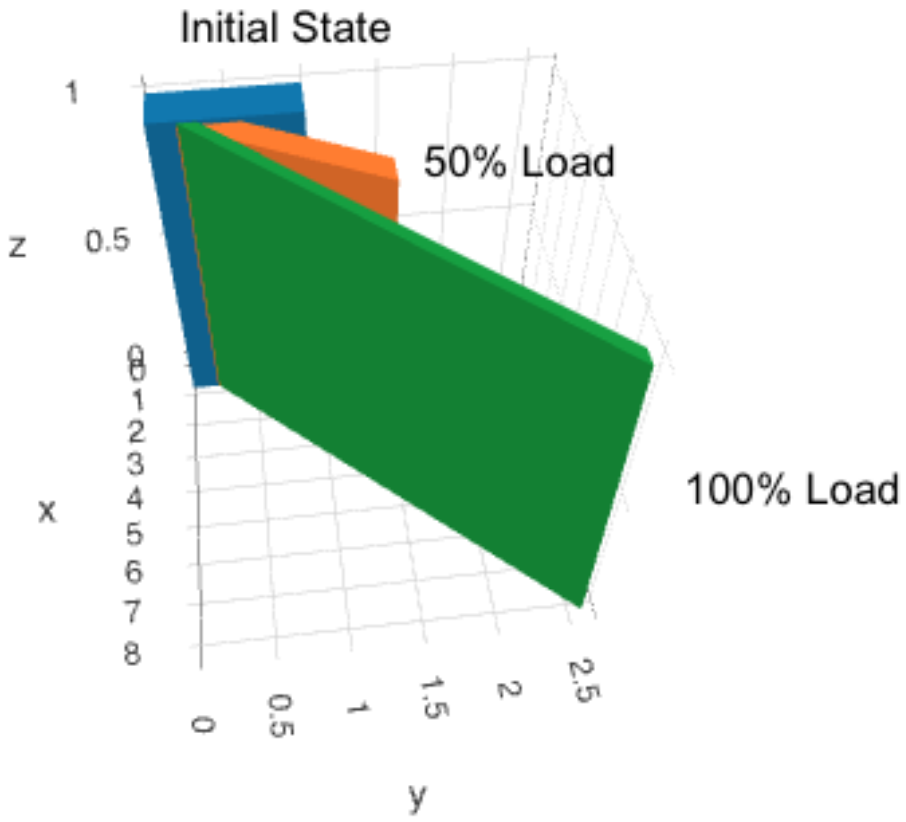

Figure 3.5

Deformation resulting from MatPointSim.py

Because all of the UMATs share the same set of parameters, a common signature file (`plasticity_t.pyf`) can be used for all of them. To debug a UMAT, the user uploads the Fortran file, changes the name of the UMAT in the Jupyter notebook (i.e. `plasticity_umat.f`), and runs the notebook. When the notebook is finished, the user can use Python to code more analysis and plots, and he/she can share these results by either pointing colleagues to the server or sending the notebook with supporting files.

CHAPTER 4

CONCLUSION

The aim of this research project was to explore the process of refactoring legacy code to keep outdated (but useful) programs relevant to the modern researcher. Fortran code written as a plugin (UMAT) for Abaqus was used to demonstrate this process. The Fortran program (`MatPointSim.f`) that called this UMAT was translated into Python, while the UMAT was compiled using f2py to avoid having to directly modify the code. `MatPointSim.py` imported the UMAT's compiled (.pyf) module and called it as a Python function. Once the code was converted to a Python program, the performance was evaluated and the program was used to generate different visualizations that would not have been practical in Fortran.

4.1 Advantages

While there are some downsides to converting the UMAT testing process to Python (such as poorer performance), the advantages far outweighed these downsides.

4.1.1 Platform-Independence

Fortran has been around for over 60 years now, and has evolved substantially over that period [25]. While considerable effort has been put forward to keep Fortran versions as

backward compatible as possible, not all Fortran compilers honor that goal and support older versions of Fortran code. Because of this, it often takes some measure of tampering with the program or compiler parameters in order to build older versions of Fortran code. This makes running something like `MatPointSim.f` on a new machine a nontrivial process. However, because f2py compiles the UMAT's Fortran code into a Python module, running `MatPointSim.py` is not a compiler-dependent or platform-dependent process. The user can simply drop the scripts onto their file system, ensure they have Python installed, and run the program. While few entry-level researchers will know how to write or compile Fortran code, almost all will have a level of familiarity with Python and will be much more willing to use a program written in Python than one in Fortran.

4.1.2 Increased Functionality

Probably the most significant advantage of using a scripting language like Python is the increased functionality it offers. Generating new results by iterating over different input variables and then plotting those results only requires a few lines of code in Python. Python also offers a wide range of builtin libraries and environments (such as the Jupyter notebooks) that provide flexibility in working with simulations and result data.

4.1.3 Adaptability

As mentioned previously, it is much easier for today's STEM researchers to read and modify scripted Python code than to edit and recompile Fortran code. As Python continues to gain steam in the scientific community [26], programs written in Python will see more sustained future use than a program written in an outdated version of Fortran. According

to a study done in 2012 of mechanical engineering programs in the United States, 20 out of 74 said they taught Fortran in a required ME course [27].

4.1.4 Collaboration

Using Jupyter notebook provides a streamlined user experience that is especially useful to researchers that are collaborating on a computing project. For the UMAT project, a Jupyter server can now be setup on a remote machine that MSU researchers can easily access. If a faculty member wants to test the validity of a new UMAT file before publishing results through Abaqus, he/she can upload the file to the remote machine and run the Jupyter notebook used for testing. The notebook can handle the conversion to Python, gathering results, and generating of the plots all in one web page, quickly and easily presenting the researcher with everything he/she needs to know if the UMAT is providing expected results. If the researcher wants to write any more custom tests for the UMAT, he/she can code new tests in Python directly in the web page and see interactive results, potentially without even having to rerun the UMAT.

4.2 Future Work

Now that the process for testing Fortran UMAT's is in Python, there are several more steps that could be taken to make the testing as streamlined and flexible as possible. By using Python GUI libraries or even continuing to use Jupyter notebooks, an interface could be created for selecting different loading scenarios and input variations to run the UMAT on. Results could be immediately presented in the form of plots or tables that would be compared with expected values. If tests on lots of variations of the inputs are desired,

the program could even be parallelized with relative ease using MPI (Message Passing Interface) and Python's mpi4py library.

The foundational work for all kinds of improvements has been accomplished. Even a researcher with just a basic understanding of Python could implement new features or modifications to how the UMATs are tested. Whatever direction the software will be taken next just depends on what kind of features or tools researchers want out of it.

REFERENCES

[1] J. E. Sammet, "Programming Languages: History and Future," *Commun. ACM*, vol. 15, no. 7, 7 1972, pp. 601–610.

[2] "TIOBE Programming Community Index Definition," 2018, Available at https://www.tiobe.com/tiobe-index/programming-languages-definition/.

[3] A. Ricadela, "Computing heads for the clouds," *Business Week*, 2007.

[4] C. Hoffa, G. Mehta, T. Freeman, E. Deelman, K. Keahey, B. Berriman, and J. Good, "On the Use of Cloud Computing for Scientific Workflows," *2008 IEEE Fourth International Conference on eScience*, 2008, pp. 640–645.

[5] K. J. Millman and M. Aivazis, "Python for Scientists and Engineers," *Computing in Science & Engineering*, vol. 13, no. 2, 2011, pp. 9–12.

[6] R. D. Mcginty, *Multiscale Representation of Polycrystalline Inelasticity*, doctoral dissertation, Georgia Institute of Technology, 2001.

[7] A. van Deursen, P. Klint, and C. Verhoef, "Research Issues in the Renovation of Legacy Systems," *Fundamental Approaches to Software Engineering*, J.-P. Finance, ed., Berlin, Heidelberg, 1999, pp. 1–21, Springer Berlin Heidelberg.

[8] J. K. Ousterhout, "Scripting: higher level programming for the 21st Century," *Computer*, vol. 31, no. 3, 1998, pp. 23–30.

[9] L. Marshall and J. Webber, "Gotos Considered Harmful and Other Programmers' Taboos.," *PPIG*, 2000, p. 14.

[10] E. W. Dijkstra, "Letters to the Editor: Go to Statement Considered Harmful," *Commun. ACM*, vol. 11, no. 3, 3 1968, pp. 147–148.

[11] A. M. Erosa and L. J. Hendren, "Taming control flow: a structured approach to eliminating goto statements," *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*, 1994, pp. 229–240.

[12] "Fortran Programming Guide," 2010, Available at https://docs.oracle.com/cd/E19957-01/805-4940/6j4m1u7qp/index.html.

[13] D. M. Ritchie, B. W. Kernighan, and M. E. Lesk, *The C programming language*, Prentice Hall Englewood Cliffs, 1988.

[14] "numpy.asfortranarray," 2018, Available at https://docs.scipy.org/doc/numpy-1.15.0/reference/generated/numpy.asfortranarray.html.

[15] P. Peterson, "f2py history and future," 2006, Available at http://pearu.blogspot.com/2006/07/f2py-history-and-future.html.

[16] P. Peterson, *F2PY: A tool for connecting Fortran and Python programs*, vol. 4, 1 2009.

[17] P. Peterson, "F2py users guide and reference manual," *Revision*, vol. 1, 2005, pp. 2001–2005.

[18] J. Aycock, "Converting Python virtual machine code to C," *Proceedings of the 7th International Python Conference*, 1998, pp. 76–78.

[19] M. Lutz, *Learning Python: Powerful Object-Oriented Programming*, " O'Reilly Media, Inc.", 2013.

[20] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. E. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. B. Hamrick, J. Grout, S. Corlay, and others, "Jupyter Notebooks-a publishing format for reproducible computational workflows.," *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, ed., 2016, pp. 87–90.

[21] N. Braun, T. Hauth, C. Pulvermacher, and M. Ritter, "An Interactive and Comprehensive Working Environment for High-Energy Physics Software with Python and Jupyter Notebooks," *Journal of Physics: Conference Series*, vol. 898, no. 7, 2017, p. 72020.

[22] "plot.ly," 2018, Available at http://plot.ly/Python.

[23] "Deformation Gradient," 2017, Available at http://www.continuummechanics.org/deformationgradient.html.

[24] R. Brannon, "F-tables for prescribed deformation," 2013, Available at https://csmbrannon.net/2013/11/02/f-tables-for-prescribed-deformation/.

[25] J. L. Overbey, S. Negara, and R. E. Johnson, "Refactoring and the Evolution of Fortran," *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, Washington, DC, USA, 2009, SECSE '09, pp. 28–34, IEEE Computer Society.

[26] F. Perez, B. E. Granger, and J. D. Hunter, "Python: An Ecosystem for Scientific Computing," *Computing in Science & Engineering*, vol. 13, no. 2, 2011, pp. 13–21.

[27] B. K. Hodge and W. G. Steele, "A Survey of Computational Paradigms in Undergraduate Mechanical Engineering Education.," *Journal of Engineering Education*, vol. 91, no. 4, 10 2002, pp. 415–417.

APPENDIX A

CODE SNIPPETS

This appendix contains some of the code snippets that were used in a Jupyter notebook to generate the plots shown throughout this thesis. Listing A.1 is essentially a driver for `MatPointSim` that repeats a loading scenario over nine different loading directions.

Listing A.1

Iterating UMAT Call

```
from MatPointSim import sim

all_stress = []
all_strain = []
all_time = []
for i in range(3):
    all_stress.append([])
    all_strain.append([])
    all_time.append([])
    for j in range(3):
        time, eff, stress, strain = sim(loading_i=i+1,
            loading_j=j+1)
        all_stress[i].append(stress[:])
        all_strain[i].append(strain[:])
        all_time[i].append(time[:])
```

The code in Listing A.2 can then be used to plot the stress over each of these loading directions.

Listing A.3 contains code that uses a Python library called plotly to visualize the deformation of the material caused by the loading scenario.

Listing A.2

Plotting Stress

```
fig, axs = plt.subplots(3,3, figsize=(15,10))
for j in range(3):
    for k in range(3):
        time = all_time[j][k]
        n = len(time)
        if not n: # Skip plots that don't have data
            axs[j,k].axis('off')
            continue

        # Normalize Stress
        stress = np.array(all_stress[j][k])
        s_min = stress.min()
        s_max = stress.max()

        for i in range(6):
            axs[j,k].scatter(time,
                [i+1,]*len(stress), s=144,
                c=[s[i] for s in stress],
                vmin=s_min, vmax=s_max,
                cmap=plt.cm.coolwarm)

        axs[j,k].set_title("Stress with loading {},{}".format(
                j+1,k+1))
        axs[j,k].set_xlabel("Time")
        axs[j,k].set_ylabel("Tensor Components")

from matplotlib.lines import Line2D
legend_items = [Line2D([0],[0], color='w'),
    Line2D([0],[0],marker='o',color='w',
    markerfacecolor=plt.cm.coolwarm(0), markersize=15),
    Line2D([0],[0],marker='o',color='w',
    markerfacecolor=plt.cm.coolwarm(1.),
    markersize=15)]

axs[1,1].legend(legend_items,
    ['Shaded to show stress intensity', 'Minimum Stress',
    'Maximum Stress'], loc="center", fontsize='large')
plt.tight_layout()
plt.show()
```

Listing A.3

Deformation Plot

```
from plotly import tools

all_data = []
for defgrd in all_dfgrd[i][j]:
  defgrd = np.array(defgrd)
  coords = zip(x,y,z)
  new_coords = [np.array(c) for c in coords]
  new_coords = [(np.dot(defgrd[0], coord), np.dot(defgrd[1],
    coord), np.dot(defgrd[2], coord))
    for coord in new_coords]
  new_coords = zip(*new_coords)

  data = [
    go.Mesh3d(
      x = new_coords[0],
      y = new_coords[1],
      z = new_coords[2],
      i = [7, 0, 0, 0, 4, 4, 6, 6, 4, 0, 3, 2],
      j = [3, 4, 1, 2, 5, 6, 5, 2, 0, 1, 6, 3],
      k = [0, 7, 2, 3, 6, 7, 1, 1, 5, 5, 7, 6],
      flatshading=True)]

  all_data.append({'data':data})

new_data = [all_data[0]['data'][0],
  all_data[len(all_data)/2]['data'][0],
  all_data[-1]['data'][0]]

layout = go.Layout(
  xaxis=go.layout.XAxis(
    title='x',
    ),
  yaxis=go.layout.YAxis(
    title='y'
  ))

fig = go.Figure(data=new_data, layout=layout)
py.offline.iplot(fig, filename='Simul_Compression_Torsion')
```