COMPUTER VISION AND SIMULATION TOOLS FOR THREE-DIMENSIONAL RANDOM ANTENNA ARRAYS

An Undergraduate Research Scholars Thesis

by

JOSHUA THOMAS RUFF

Submitted to the Undergraduate Research Scholars Thesis program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Gregory Huff

May 2017

Major: Electrical Engineering

ABSTRACT

Computer Vision and Simulation Tools For Three-Dimensional Random Antenna Arrays

Joshua Thomas Ruff
Department of Electrical Engineering
Texas A&M University

Research Advisor: Dr. Gregory Huff
Department of Department of Electrical and Computer Engineering
Texas A&M University

This research project aims to develop and improve auxiliary tools which aid in ongoing research on beamforming with random antenna arrays. One of the challenges of conducting research on random arrays is the tedious nature of preparing electromagnetic simulations to test beamforming algorithms. This project presents an automation framework written in Python which will expedite the setup of simulations and reduce the room for error during this process. A computer vision system designed to locate the feedline of the patch antennas used in this array is used in the random array lab setup. Refinements to the light filtering software and the position finding software are presented, and a machine learning approach for distinguishing between different antennas is explored.

ACKNOWLEDGMENTS

I would like to thank Dr. Gregory H. Huff and all other members of the Huff Research Group who provided help, advice, and guidance throughout this process. I would also like to thank Deanna Sessions and Jacob Freking for taking the time to edit the drafts of this thesis.

CONTRIBUTORS

The Medusa system was developed in the Huff Research Group Labratory over the course of several years. Modifications to the computer vision system and the development of the dual quaternion interface for HFSS were done in close collaboration with Jacob Freking, who converted the code in Appendix D and Appendix E from Python2 to Python 3. Steven Yeh provided test cases for the random array generator.

All other work conducted for this thesis was completed by the student independently.

NOMENCLATURE

HRG	Huff Research Group
HFSS	Ansys HFSS, a High Frequency Electromagnetic Simulation Software
Phased Array	An antenna array with phased inputs
UAV	Unmanned Aerial Vehicles
WSN	Wireless Sensor Network
CAD	Computer Aided Design
API	Application Program Interface

TABLE OF CONTENTS

Page

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

1.1 Random Phased Arrays

Random phased arrays show a variety of advantages over more traditional antenna array geometries. Random arrays allow for sidelobe reduction without amplitude tapering[1][2][3]. The development of random antenna arrays is critical to the maturation of distributed beamforming techniques which can be used for less conventional mutable communication networks[4]. Random arrays provide the opportunity to bring phased array beamforming to nontraditional platforms such as phased arrays on networks of satellites[2], phased arrays on morphing formations of unmanned aerial vehicles (UAVs)[1], and grounded wireless sensor networks (WSNs)[1][4]. An analytical treatment of the problem of volumetric spherical random antenna arrays was provided by Buchanan[1]. The development of beamforming algorithms in volumetric random arrays also

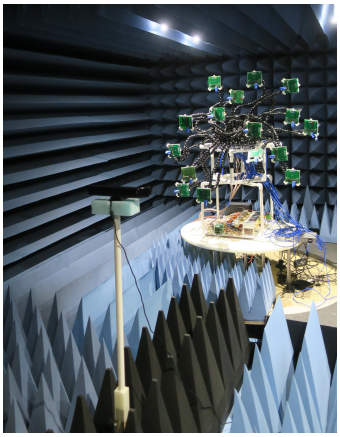

Figure 1.1: Random Array with Kinect Computer Vision Setup

presents unique challenges which were not present during the development of classical phased arrays. The exact location of each antenna is known upon the construction of the array in a traditional array. In a random array, the exact location of each antenna in a 3-dimensional space needs to be

actively measured to apply phasing as well as to replicate the geometry in simulation[5]; tasks which are made more challenging by the entropy innate in random arrays.

1.2 Computer Vision System

One method with which antenna location can be determined is by use of computer vision systems[6]. Computer vision systems vary in sophistication but have the advantage of being faster than measuring element placement by hand; however, they bring their own set of challenges to the problem. The current setup in the Huff Research Group lab for measuring antenna position (shown in Figure 1.1) involves using an XBOX Kinect to locate the feed point. First, light from the LEDs positioned at known distances from each antenna's feed point is filtered (Figure 1.2 Left). Next, boxes are drawn around sharp contours in the filtered image (Figure 1.2 Center). The location algorithm looks for two larger recognition rectangles, each completely containing two smaller location rectangles (implying no overlap). When this case exists, the antenna is successfully recognized by the software. The center coordinates of each of the location rectangles are combined with prior knowledge of the position of each LED to approximate the feed point position(Figure 1.2 Right).

Figure 1.2: Computer Vision System: (left to right) Patch Antenna with LEDs Filtered, Boxes Drawn Around Sharp Filtering Contours, Successful Feed Point Location

1.2.1 Limitations of Current Computer Vision System

There are a number of limitations to the computer vision system in its current state. Ambient light can significantly reduce the effectiveness of this system which relies on sharp changes

in color and intensity to locate objects. Potential for improvement of this system is found in the implementation of more sophisticated computer vision algorithms and filtering techniques to the measurement software. The existing lab setup has additional limitations. Firstly, certain rotations will cause the antenna recognition system by to malfunction by making the recognition rectangles to overlap (Figure 1.3). Another limitation is the ability to only measure one antenna at a

Figure 1.3: Antenna recognition system failure caused by overlapping boxes

time, and one type of antenna at a time. With each additional antenna measured simultaneously, the complexity of a potential location algorithm increases significantly. Mixing in different types of antennas, each with its own pattern of marking LEDs, also compounds the problem. In this vein, the application of machine learning techniques with computer vision offers a promising alternative to the existing approach to element location finding, and thus will be investigated for this application.

1.3 Electromagnetic Simulations

Modeling and simulating a system in Ansys HFSS, an electromagnetics CADing software, is a critical part of the design process for any electromagnetics project. Once a lab setup has been measured, the orientations, magnitude and phase tapering, and positions for each element of the

random array must be entered into HFSS by hand; a process which becomes tedious and time consuming as the number elements in the array increase. The repetetiveness of specifying large numbers of array elements creates a situation where automated drawing is advantageous .

1.3.1 Goals of an Automation Framework for HFSS

The first goal of any automation framework for HFSS would be to speed up the process of modeling random arrays. Currently, the process for specifying a random array can take several hours depending on the number of elements, and with an automation framework this could drop down to a few minutes. The speed and convenience with which this is created can then be leveraged to create other features. Positions and orientations measured through the computer vision system would be able to be quickly and automatically replicated in HFSS. This could also be used to automate the process of randomly generating large numbers of arrays, simulating them in HFSS, and stripping out the desired results. Another application of an automation framework is the use of alternative coordinate systems. HFSS can only specify drawings in cartesian coordinate systems, while Quaternions and Dual Quaternions are often used for the sorts of aerial vehicle systems random arrays will be applied to. For testing puproses, it is then desirable to be able to 'sample' the Dual Quaternion positions and orientations of the vehicles, then specify and simulate each sampled configuration in HFSS. Longer term goals include expanding the HFSS native capabilities through the implementation of functions capable of drawing fractal structures or more challenging parametric structures than can be handled by the native parametric drawing capability

1.3.2 Existing Scripting Capabilities of HFSS

A Visual Basic API for HFSS exists, as does a method to call functions from this API through Python code; however, the syntax for this is unnatural and results in code which stylistically is quite different from Python. As an example take the following code sample which shows the native environment's CreateSphere method.

```
oEditor.CreateSphere(
   [
```

```
    "NAME:SphereParameters",
    "XCenter:="   , "0mm",
    "YCenter:="   , "0mm",
    "ZCenter:="   , "0mm",
    "Radius:="    , "1.01980390271856mm"
  ],
  [
    "NAME:Attributes",
    "Name:="   , "Sphere1",
    "Flags:="   , "",
    "Color:="   , "(132 132 193)",
    "Transparency:=" , 0,
    "PartCoordinateSystem:=", "Global",
    "UDMId:="   , "",
    "MaterialValue:=" , "\"vacuum\"",
    "SolveInside:="   , True
  ])
```

Numerical values for the sphere's coordinates and radius have to be concatenated with their respective units, then passed to the HFSS environment as part of an array containing other string literals. This is the case for all other drawing functions, and leads to a situation where it would be advantageous to build a layer of abstraction on top of the HFSS API to minimize syntax errors. Such a library could also provide for the use of design variables to allow models to be slightly modified after construction, and facilitate the drawing of devices in local coordinate systems.

1.4 Dual-Quaternion Coordinate System Transformation

HFSS posesses a limited method of transforming coordinate systems natively. Relative coordinate systems can be specified by a translation of the origin and directional vectors of the X-axis and Y-axis. The directional vectors together encompass all the information from any rotations per-

formed on the coordinate system relative to the origin. This means that when simulating an array or some other structure defined using an alternative coordinate system, calculations to model this in HFSS must be done and entered by hand, which, in turn, becomes another natural way to extend the automation framework.

Quaternion and dual quaternion representations have a number of advantages over traditional 3x3 rotation matrices. Quaternion multiplication requires fewer arithmetic operations than 3x3 matrices[7]. Quaternion products are easily renormalized, whereas the nearest orthonormal matrix to the result of 3x3 matrix multiplication is not so easily found. These computational efficiencies are an advantage of quaternions over more traditional representations. Quaternions also avoid some singularities which plague 3x3 matrix representations such as gimbal lock[8]. Because of these features, quaternion systems find wide use in robotics, computer graphics, aircraft, and spacecraft navigation[7]. Dual-quaternions are needed to represent both a translation and rotation. Consequently, one focus of this paper will be the theory behind and methods necessary to represent a dual-quaternion system in HFSS.

1.4.1 Complex Numbers and 2D Rotation Matrices

Quaternion coordinate systems use complex numbers as a method for representing rotations. McDonald's paper [9] draws an analogue between complex numbers and rotation matrices which is very useful for developing an intuition for quaternion transformations. He begins with the subset of 2x2 scale matrices of the form:

$$A = \begin{bmatrix} a & 0 \\ 0 & a \end{bmatrix}$$

in which matrix addition and subtraction behave the same as real number addition and subtraction. These matrices function as an analogue for real numbers.

Next he introduces the matrix :

$$I = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} \cos(90^\circ) & -\sin(90^\circ) \\ \sin(90^\circ) & \cos(90^\circ) \end{bmatrix}$$

What makes I unique is I^2 equals the scale matrix $[-1]$. Where in normal complex numbers, $i = \sqrt{-1}$ wasn't really something that existed, in matrix form the square root of negative one exists in the form of I. This implies that a unit complex number $cos\theta + jsin\theta$ equals:

$$R = \begin{bmatrix} \cos\theta & 0 \\ 0 & \cos\theta \end{bmatrix} + \begin{bmatrix} 0 & -\sin\theta \\ \sin\theta & 0 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{1.1}$$

One of the subtleties of quaternion notation is the fact that every vector is a complex number[9]. Purely real vector notation would show matrix multiplication as

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

However, if the vector is interpreted as a complex number, multiplication by the rotation matrix would be notated as:

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x & -y \\ y & x \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta & -y\cos\theta + x\sin\theta \\ y\cos\theta + x\sin\theta & x\cos\theta - y\sin\theta \end{bmatrix} \tag{1.2}$$

The significance is that both methods of representing vectors give the same result:

$$(x\cos\theta - y\sin\theta) + i(y\cos\theta + x\sin\theta)$$

1.4.2 Quaternions

Quaternions are formally defined as an extension of complex numbers[7]:

$$q = w + ix + jy + zk \tag{1.3}$$

where i, j, and k are orthogonal imaginary unit vectors under the constraints:

$$\begin{gathered} i^2 = j^2 = k^2 = -1 \\ ij = k, \quad jk = i, \quad ki = j \\ ji = -k, \quad kj = -i, \quad ki = -j \end{gathered} \tag{1.4}$$

These rules are made more intuitive by McDonald's method[9] of building a 3D analogue of the relationship between complex numbers and rotations in a 2D space. Because quaternions consist of 4 elements, a 4x4 matrix is necessary to fully represent them. Similar to the 2x2 case, real numbers are represented by 4x4 matrices of the form:

$$[A] = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & a & 0 \\ 0 & 0 & 0 & a \end{bmatrix}$$

There are multiple solutions to $\sqrt{[-1]}$ in 4x4 matrices. The following three are chosen to yield a matrix representation of quaternions which most closely follows the 2x2 analogue:

$$i = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix} \quad j = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} \quad k = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{1.5}$$

The rules defined in (1.4) can be derived directly from multiplying the imaginary matrices in (1.5). Furthermore, these matrix definitions can be combined with (1.3) to yield a quaternion fully em-

bedded into a 4x4 matrix:

$$q = \begin{bmatrix} w & -z & y & x \\ z & w & -x & y \\ -y & x & w & z \\ -x & -y & -z & w \end{bmatrix} \tag{1.6}$$

Another notation for quaternions is useful for describing certain properties[10]:

$$q =< w, \vec{u} > \quad where \quad \vec{u} = [x, y, z]^T \tag{1.7}$$

With this, quaternion addition and multiplication by scalars can be defined.[7]

$$aq_1 + bq_2 = bq_2 + aq_1 =< aw_1 + bw_2, a\vec{u_1} + b\vec{u_2} >$$

Quaternion multiplication can also be defined, but like matrix multiplication is not commutative

$$q_1 q_2 =< w_1 w_2 - \vec{u_1} \cdot \vec{u_2}, w_1\vec{u_2} + w_2\vec{u_1} + \vec{u_1} \times \vec{u_2} >$$
$$q_2 q_1 =< w_1 w_2 - \vec{u_1} \cdot \vec{u_2}, w_1\vec{u_2} + w_2\vec{u_1} - \vec{u_1} \times \vec{u_2} >$$

The conjugate of a quaternion is found similar to complex conjugates, by negating every complex term:

$$q^* = w - ix - jy - kz \tag{1.8}$$

By plugging the conjugate definition from (1.8) into the matrix form of a quaternion (1.6), it becomes evident that for unit quaternions, $q^* = q^T$. Finally, with the definition of q^*, the norm square and thus inverse quaternion can be defined:

$$||q||^2 = qq^* = w^2 + x^2 + y^2 + z^2 = q \cdot q \tag{1.9}$$

$$q^{-1} = (1/||q||^2)q^* = q^* \quad for \quad ||q||^2 = 1 \tag{1.10}$$

1.4.3 Unit Quaternions and Rotations

The intutitive way to define quaternion rotation is now to directly extend the complex rotation matrix method $R\vec{v}$ into three dimensions. Before this can be attempted, $\vec{v}$ needs to be expressed in quaternion notation, which is accomplished by assigning a w component of 0: $q_v =< 0, \vec{v} >$.

Quaternions with unit magnitude are used to represent rotations in three dimensions to leverage the fact that $q^{-1} = q^*$. A rotation of θ around the axis in the direction of $\hat{u}$ is represented in unit quaternion notation as:

$$q_u =< \cos\theta, \sin(\theta)\vec{u} > \tag{1.11}$$

As an example, (1.12) shows a quaternion rotation by θ around the z axis, and extends this to the matrix notation introduced in (1.6):

$$q_z =< \cos\theta, \sin\theta[0,0,1] >= \cos\theta + \sin\theta k \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \\ 0 & 0 & -\sin\theta & \cos\theta \end{bmatrix} \tag{1.12}$$

The matrix in (1.12) shows how the 2-D rotation matrix from (1.1) is now embedded into the top left corner of the quaternion representation of the transformation. The particular choices of the i, j, and k matrix allowed for this to occur. The bottom right corner the quaternion contains the information for a rotation by $-\theta$, which will represent an extra rotation in the $w - z$ plane. This is where the limitation of the rotation matrix analogue breaks down and can clearly be shown by applying this rotation to $\vec{v} = [1, 0, 1]$:

$$\begin{aligned} \vec{v'} = q_z\vec{v} = (\cos\theta + k\sin\theta)(i + k) = i\cos\theta + k\cos\theta + ki\sin\theta + k^2\sin\theta \\ = -\sin\theta + i\cos\theta + k\cos\theta + j\sin\theta =< -\sin\theta, [\cos\theta, \sin\theta, \cos\theta] > \end{aligned} \tag{1.13}$$

compare this with the expected result:

$$\vec{v'} =< 0, [\cos\theta, \sin\theta, 1] > \tag{1.14}$$

The x and y components of the operation in (1.13) were rotated corectly. However, the rotation in the w-z plane reduced the value of the z coordinate, and added a w component to what was supposed to be only a vector. This is a limitation of building directly on the rotation matrix analogue, the method of rotation will need to be modified from this.

To show where the modification comes from, McDonald first starts by reversing the order of the rotation:

$$\begin{aligned} \vec{v'} = \vec{v}q_z = (i+k)(\cos\theta + \sin\theta k) = i\cos\theta + k\cos\theta + ik\sin\theta + k^2\sin\theta \\ = i\cos\theta + k\cos\theta - j\sin\theta - \sin\theta =< -\sin\theta, [\cos\theta, -\sin\theta, \cos\theta] > \end{aligned} \tag{1.15}$$

This result reversed the direction of the desired rotation about the z axis, but the w-z plane rotation remains the same. To rectify this, the same operation is performed with q_z^{-1}.

$$\begin{aligned} \vec{v'} = \vec{v}q_z^{-1} = (i+k)(\cos\theta - \sin\theta k) = i\cos\theta + k\cos\theta + ik\sin\theta - k^2\sin\theta \\ = i\cos\theta + k\cos\theta + j\sin\theta + \sin\theta =< \sin\theta, [\cos\theta, \sin\theta, \cos\theta] > \end{aligned} \tag{1.16}$$

Now the rotation about the z axis is correct, and the w-z plane rotation is reversed from that of (1.13). The final result of all this is quaternion rotations are performed with two operations on $\vec{v}$:

$$\vec{v'} = q_z\vec{v}q_z^{-1} \tag{1.17}$$

The rotation in (1.16) now rotates v by θ about the z axis twice, while the opposing w-z rotations from q_z and q_z^{-1} cancel with one another. Thus (1.18) is the final form of quaternion rotation about axis $\vec{u}$, with the definition of qu from (1.11) modified to eliminate the double rotation about the z

axis:

$$\boxed{\begin{array}{c}\vec{v'} = q_u\vec{v}q_u^{-1} = q_u\vec{v}q_u^* \\ q_u =< \cos(\theta/2), \sin(\theta/2)\vec{u} >\end{array}} \tag{1.18}$$

Rotations can also be composed from multiplications of multiple quaternions[7]:

$$v^{'} = (pq)v(pq)^* \tag{1.19}$$

This fully specifies rotations around any axis. However, to fully specify translations as well as rotations, quaternions need to be extended to a dual number system.

1.4.4 Dual Numbers

Goddard provides the background on dual numbers in his dissertation[7]. He begins with the definition of a dual number:

$$d = a + \epsilon b \quad where \quad \epsilon^2 = 0 \tag{1.20}$$

He then defines a variety of operations for dual numbers. Note the similarity to complex number operations.

- Addition:

$$d_1 + d_2 = (a_1 + \epsilon b_1) + (a_2 + \epsilon b_2) = a_1 + a_2 + \epsilon(b_1 + b_2) \tag{1.21}$$

- Multiplication:

$$d_1 d_2 = (a_1 + \epsilon b_1)(a_2 + \epsilon b_2) = a_1 a_2 + \epsilon(a_1 b_2 + b_1 a_2) \tag{1.22}$$

- Conjugation:

$$d^* = a - \epsilon b \tag{1.23}$$

- Modulus:

$$|d| = a \tag{1.24}$$

- Conjugate multiplication:

$$dd^* = a^2 \tag{1.25}$$

- Finally, he defines a dual angle:

$$\hat{\theta} = \theta + \epsilon d \tag{1.26}$$

where θ is the angle between the line vectors and d is the shortest distance between the lines[7]

1.4.5 Dual Quaternions

Next, Goddard builds upon the definition of dual numbers by defining the Dual Quaternion.

$$\hat{q} = r + \epsilon s \tag{1.27}$$

where r and s are quaternions. Again he defines several operations for Dual Quaternions. Dual quaternion multiplication has definitions similar to quaternion multiplication and addition:

$$\hat{q}_1 + \hat{q}_2 = r_1 + r_2 + \epsilon(s_1 + s_2) \tag{1.28}$$

$$\hat{q}_1\hat{q}_2 = r_1r_2 + \epsilon(s_1r_2 + r_1s_2) \tag{1.29}$$

In addition to this, he defines conjugation, square magnitude, and inverse of a dual quaternion:

$$\hat{q}^* = r^* + \epsilon s^* \tag{1.30}$$

$$||\hat{q}||^2 = \hat{q}\hat{q}^* = rr^* + \epsilon(rs^* + sr^*) \tag{1.31}$$

$$\hat{q}^{-1} = \hat{q}^*/||\hat{q}||^2 \tag{1.32}$$

Finally, he defines a unit dual quaternion as having:

$$||\hat{q}||^2 = 1 + \epsilon 0 \tag{1.33}$$

This implies:

$$rr^* = 1 \quad and \quad rs^* + sr^* = 0 \tag{1.34}$$

1.4.6 Representing Both a 3-D Rotation and a Translation with Dual Quaternions

With the definition of Dual Quaternions complete, they can now be used to fully represent 3D rotations and translations. Let t be a quaternion vector representing a translation.

$$t = 0 + ix + jy + kz \tag{1.35}$$

q is a quaternion rotation about some axis $\vec{u}$ as defined in (1.18). A dual quaternion representation of these is then:

$$\boxed{\hat{q} = q + \epsilon \frac{t}{2} q} \tag{1.36}$$

2. METHODS

The challenges of working with random arrays here are approached from two different angles. Firstly, an automation framework is develeoped along the lines of the goals laid out in the introduction, and serves to automate the generation of large HFSS simulation models. Secondly, the computer vision system is modified to improve the ability to measure antenna position in a lab setting, and allow for various tests of beamforming with Medusa, a random antenna array test setup.

2.1 Electromagnetic Simulations

The current lab setup uses a sixteen-element antenna array to test beamforming techniques, the entirety of which needs to be modeled accurately in HFSS to verify test results with simulation results (Figure 2.1). Because of the challenges innate constructing the model, automation with Python becomes a viable solution. HFSS has a feature where the user can construct or modify a part of a model through the GUI editor, then record this sequence of actions to a Python file. This process outputs code of the form similar to the $CreateSphere$ method discussed in the introduction. Without modification this is a rudimentary form of automation, but its usefulness is hindered by the syntax.

2.1.1 Abstraction Layers in the HFSS Automation Framework

Code which relies upon directly on the native HFSS API quickly becomes a challenge to write and debug. Another drawback of the native scripting environment is every value used to specify the geometry of the antenna array is hardcoded into the strings. Hardcoded values limit the reusability and, as a result, the overall usefulness of the script. To compensate for these weaknesses, an automation library can be split into three distinct and (mostly) isolated layers within the code, shown in Figure 2.2. These abstraction layers help to facilitate the writing of clean, functional, pythonic code while automating HFSS.

The lowest layer consists of the functions that interact directly with HFSS. This $application$

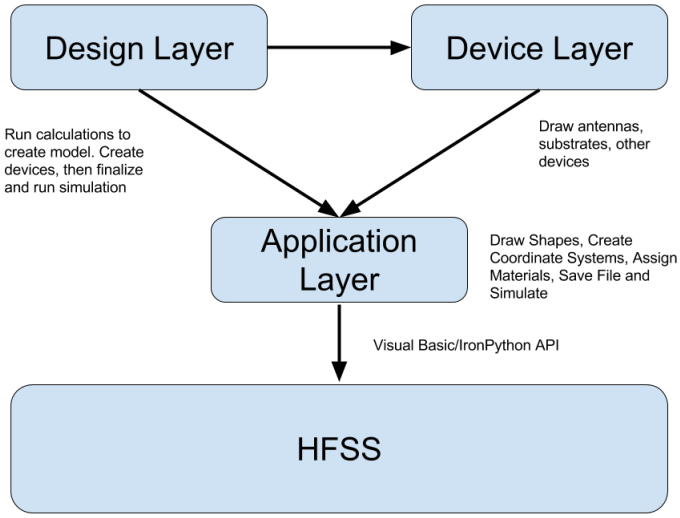

Figure 2.1: Abstraction Diagram

$layer$ includes drawing functions, functions to create relative coordinate system, assign materials to objects, assign excitations, create radiation boundaries, and handle file management. The user of the automation library can thereby be given access to any and all features of HFSS through the interface of this lowest layer.

The second layer is the $device\ layer$. As its name implies, the device layer provides a series of functions which can be used to specify various electromagnetic devices through their dimensions and constitutive materials. Device layer functions will then perform all necessary calculations using those dimensions, and call all the application layer functions necesary to draw the device model in HFSS. Device layer functions will then return the name of all objects needed in the design layer to perform operations such as subtracting out the radiation boundary or assigning excitations to waveports.

The third and nominally the highest abstraction layer in the HFSS library is then the $design\ layer$. Design layer files are the main files for this automation framework. Functioning in this role, design layer files will import the coordinates and rotations of devices in the design under test, call any device layer functions necessary to draw those devices in the program, and call any

application layer functions necessary to handle the assignment of boundary conditions, excitations, and wave fronts. Additionally, the design layer will call application layer functions to handle file management and simulation initialization. In this manner, the organization provided by this abstraction works to facilitate the writing of automation scripts.

2.1.2 Application Layer

Many of the application layer functions can be classified into a few major categories. Drawing functions specify 2-D or 3-D shapes in the software. Modification functions subtract objects from one another, assign boundary conditions or waveports to geometries, and assign materials to the geometry. More general functions can assign excitations, edit simulation paramters, automate the processing of results, and automate running the simulation. All of this funcitonality is contained in the file $HFSSLibrary.py$.

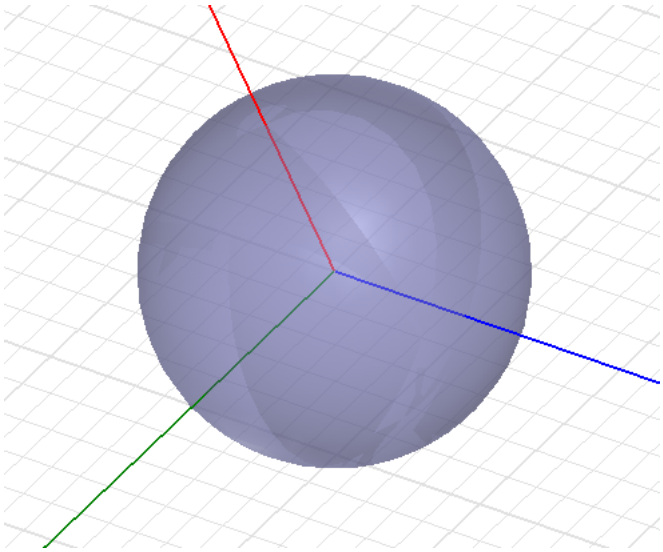

Figure 2.2: HFSS Sphere

Drawing functions all have similar functionality, and can be illustrated through the example of the drawSphere() function. A sphere, shown in HFSS in figure 2.3, can be completely specified by its positional coordinates and radius.

```
def drawSphere(oDesign, center_x, center_y, center_z, radius, units,
   material, cs, names, Transparency):
```

As per the function header shown above, the application layer function then takes in these numerical arguments, along with a single string for their units, and handle all the concatenation and other string operations necessitated by the HFSS API. It also handles choosing a coordinate system, naming the object in HFSS, and assigning a transparency value to the shape. This allows the user of the automation library to perform any calculations in a higher level of abstraction (the design layer or device layer), then pass these values to the application layer, where all the string manipulation is handled automatically. Every drawing function called in HFSS, including but not exclusive to drawing rectangles, boxes, circles, and cylinders, can be handled in a similar manner to $drawSphere$, so that the user of the library doesn't have to touch the Visual Basic API directly.

Another important feature of the Application layer involves the creation of local coordinate systems for each device. These relative coordinate systems can be specified with a translation and rotation by the vector direction of the x and y axes relative to the global origin. This is shown through the function header below:

```
def dualQuaternionCS(oDesign,dq,units,name):
```

When a relative coordinate system is active, all operations are performed in that local coordinate system. This cuts down on the number of operations needed to draw a geometry in HFSS by eliminating the need for a separate translation and rotation for each object as they are automatically drawn in position. The drawback to this native relative coordinate system is specifying rotation's through this mehtod is not very intuitive, more common methods would be rotation matrices or quaternion rotations. To facilitate this, the application layer library provides three auxiliary coordinate system functions: $globalCS$, $rotatedCS$ and $dualQuaternionCS$. The $globalCS$ function resets the active coordinate system to be the global coordinate system. This is done in between specifying local coordinate systems for other devices, so that all coordinates are specified relative

to the same origin.

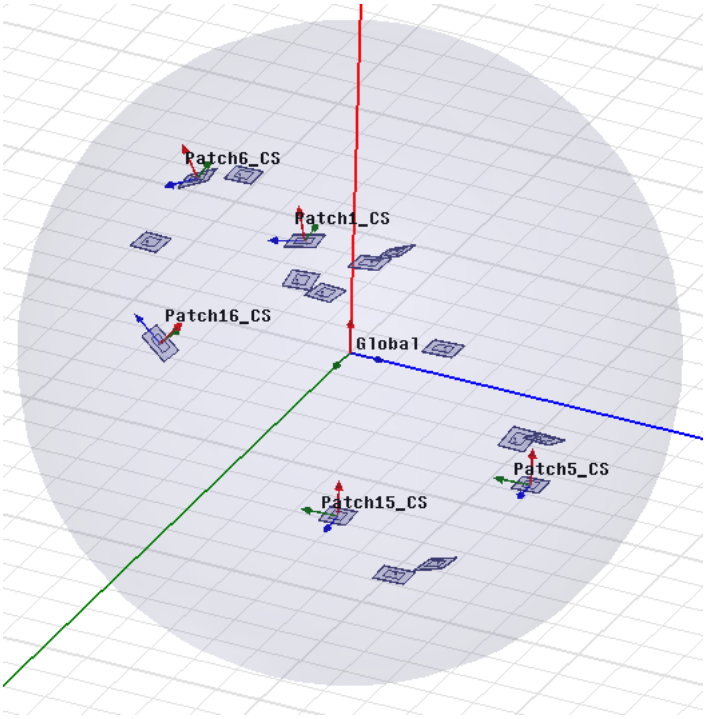

Figure 2.3: This HFSS model shows local coordinate systems and the global origin.

$rotatedCS$ takes in the same translation coordinates $createRelativeCS$ does, but the rotation is described by the angle in degrees rotated about the x-axis, y-axis, and z-axis.

```
def rotatedCS(oDesign, X, Y, Z, theta_x, theta_y, theta_z, units,
    name):
```

These are combined into a rotation matrix and multiplied by unit vectors to get the direction vectors necessary to call $createRelativeCS$. Rotations and translation measured with the Medusa computer vision system are currently output in a format that makes this function convenient.

To implement $dualQuaternionCS$ in this library, an existing open source dualQuaternion class[11] had to be converted from Python 2 to Python 3. The HFSS Library function then takes in a dualQuaternion object as a parameter, and extracts the x,y,z coordinates and rotation matrix from

the quaternion object necessary to specify the coordinate system in HFSS.

```
def dualQuaternionCS(oDesign,dq,units,name):
```

The dual quaternion is converted to cartesian specification using methods native to the dualQuaternion class. This enables numerical method programs using quaternion systems to interface with HFSS automatically.

The application layer also handles the assignment of boundary conditions, and the assignment of materials to objects. $assignFaceBoundaryMaterial$, $assignBoundaryMaterial$, and $AssignRadiationBoundary$ can all assign different geometries to be perfect conductor, infinite far field, perfect magnetic or other types of boundaries. Most objects are assigned materials as a parameter of their drawing functions. The application layer also offers access to the HFSS functionality to create custom materials through the $newProperty$, $getProperty$, and $changeProperty$ functions.

Another set of functions in the application layer handles excitations, simulation parameters and file management. $assignExcitation$ puts a wave port onto an object, then $edit\_sources$ edits the phasing and amplitude of different excitations. $insert\_setup$ allows for control of simulation parameters, and $LinearFrequencySweep$ allows for the confiugration of frequency sweeps as its name implies.

2.1.3 Device Layer

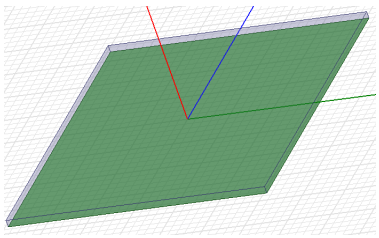

Figure 2.4: Electrically permittive substrate with copper ground plane (green)

The device layer consists of all the functions contained in $EmagDevices.py$. These functions are written to use all the application layer functions and automate the drawing of different devices in HFSS. Parameters mostly consist of the dimensions of the device, and the names of objects will be returned as necessary. Functions to automate the drawing of coax cable, copper plated substrate, and patch antennas are all currently written. This can then easily be extended to include dipole and monopole antennas, as well as other types of antennas that may be useful in random array research.

The $substrate$ function takes in the xyz coordinates of the center of the substrate, as well as a string for the material. $substrate$ then calls drawing functions, and assigns a conductive copper boundary to the bottom face of the substrate for the ground plane. The header of the function is shown below, and a screenshot of the resulting device is shown in Figure 2.4.

```
def substrate(oDesign, subX, subY, subZ, units, material,cs, name):
```

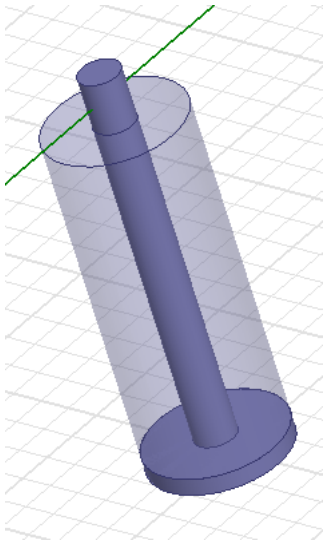

Figure 2.5: Coax Feedline for Patch Antennas

The $coax\_50\_Ohm$ function draws the coax cable shown in figure 2.5, with the xy coordinates of the center of the coax, as well as the length and substrate height as parameters. It uses the application layer functions to $drawCylenders$ and perform binary subtractions of overlapping shapes

to draw the coax, and assigns copper boundaries to the inner and outer conductors. Finally, it calls $assignExcitation$ to create a waveport on one end of the coax, and draws a perfect electrical conducting cap on the other side of this waveport.

```
def coax_50_Ohm(oDesign, center_x, center_y, length, substrate_height,
    cs, name):
```

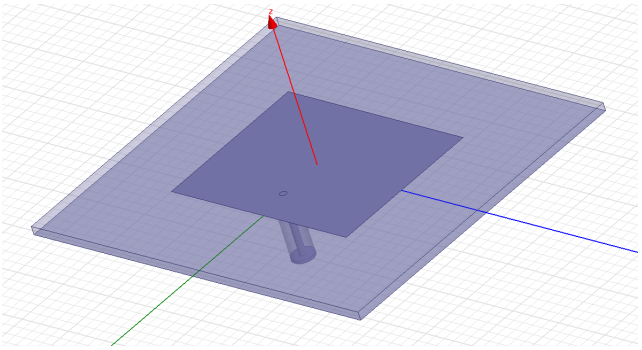

Figure 2.6: Fully drawn rectangular patch antenna, with feedline matched

The $rectangular\_patch$ function is used to draw a patch antenna with fully known dimensions in HFSS. Necessary parameters include the substrate material, length, width, and height, the patch length and width, and the probe xy location. This function calls $substrate$ and $coax\_50\_Ohm$ to place those in the proper positions relative to the patch. In doing so, the names of the devices and excitation port are returned. An example of the patch antenna is shown in Figure 2.6.

```
def rectangular_patch(oDesign, patch_length, patch_width, probe_x,
    probe_y, substrate_length, substrate_width, substrate_height,
    substrate_material,units,cs, name):
```

Another useful function is the $design\_rectangular\_patch$ function which designs a patch antenna based on the operating frequency, feedline impedance, and substrate material and height.

```
def design_rectangular_patch(oDesign, operation_frequency,
    feedline_impedance, substrate_height, substrate_permittivity,
    substrate_material, units, cs, name):
```

2.1.4 Design Layer

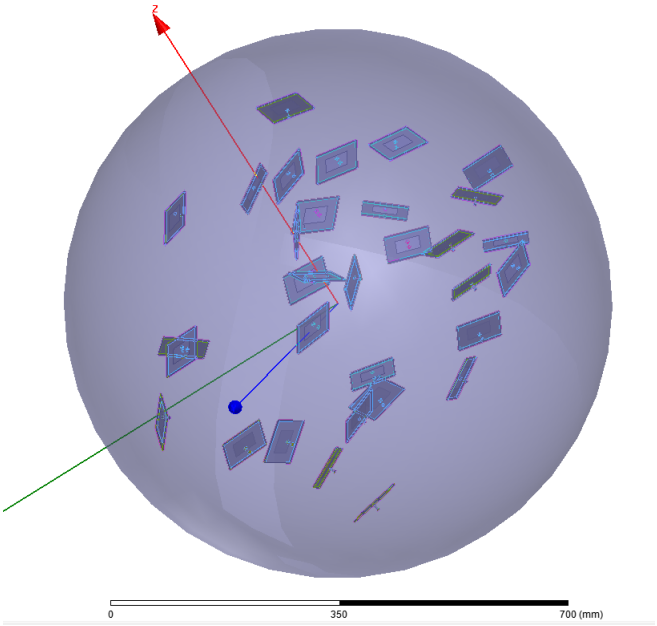

Figure 2.7: Resulting 32 Element Random array

The *Design Layer* consists of the main file for any automation using this library. Because of the design choices made in writing the application layer and device layer functions, a full array of 32 patch antennas can be specified in just a 10 line for-loop, provided the antenna positions and rotations are already imported, and the patch dimensions are already specified.

```
for i in range(0, len(positions)):
  globalCS(oDesign) #relative CSs are created based off global CS
  name = "Patch%d" % (i + 1)
  csName = name + "_CS"
```

```
  rotatedCS(oDesign, positions[i, 0], positions[i, 1], positions[i,
     2], rotations[i, 0], rotations[i, 1],rotations[i, 2]-90, "mm",
     csName)
  [temp_excitation, temp_object_names] = rectangular_patch(oDesign,
     patchL, patchW, probeX, probeY, subL, subW, subH,"FR4_epoxy",
     "mm", csName, name)
  excitations.append(temp_excitation)
  object_names += temp_object_names
```

At the beginning of each loop iteration, the active coordinate system is reset to the global origin, so that all translations and rotations are performed relative to this system instead of relative to each consecutive local coordinate system. The next few lines demonstrate another useful ability of this style of automation: objects in HFSS can be dynamically named. For example, the first patch would be named 'Patch1', its coordinate system 'Patch1_CS', its substrate 'Patch1_Substrate', and so on. This is extremely useful when editing the generated model manually after its creation.

After creating the names, the relative coordinate system is created with a call to $rotatedCS$, and the patch is drawn with a call to $rectangular\_patch$. Finally, object names and excitation names are appended to appropriate lists to allow for editing of excitaitons and subtraction of objects from the radiation boundary sphere outside the loop.

Following this loop, the radius of the radiation boundary is calculated in a method specific to the particular setup, and then simulation parameters are set up.

```
edit_sources(oDesign, excitations, modes, amplitudes, phases, "dBm",
   "deg")
globalCS(oDesign)
drawSphere(oDesign, 0, 0, 0, max_r+wavelength, "mm", "vacuum",
   "Global", "radiation_boundary", .55)
binarySubtraction(oDesign, "radiation_boundary", object_names,True)
insertSetup(oDesign, 2.45e9,"Test_Setup")
```

```
LinearFrequencySweep(oDesign,2e9,4e9, .01e7, "Test_Setup",
    "Test_Sweep")
AssignRadiationBoundary(oDesign, "radiation_boundary",
    "radiation_boundary")
```

First, the phases and amplitudes of the inputs are specified with $edit\_sources$. Next, the global origin is designated the active coordinate system so that the radiation boundary is drawn centered at the origin. Following this, the radiation boundary is drawn, and the list of objects from the array generation loop is passed to $binary\_subtraction$ to subtract them from the vaccum in the radiation boundary sphere. The simulation parameters are then set up, the radiation boundary is assigned to the outer face of the sphere, and finally, the project is saved and simulated. Thus, in fewer than 20 lines of code, (excluding the declaration of lists of positions and patch sizes) a large random array such as that shown in Figure 2.7 can automatically be specified in HFSS.

2.2 Computer Vision System

2.2.1 Fixing Rotations

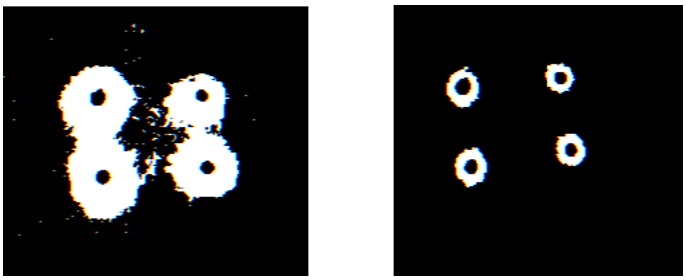

Figure 2.8: Halos from filtering around LEDs. The old filter value is on the left, while the new filter is on the right

The most significant major issue with the computer vision system was the inability to measure a rotated antenna. The Medusa software recognized when it had found an antenna by counting the number and arrangement of rectangles. Overlapping rectangles, shown in the introduction

in Figure 1.3, signal to the computer that the contours around the LEDs are not clear enough to distinguish the antenna from the surroundings. The approach to this problem was to change the

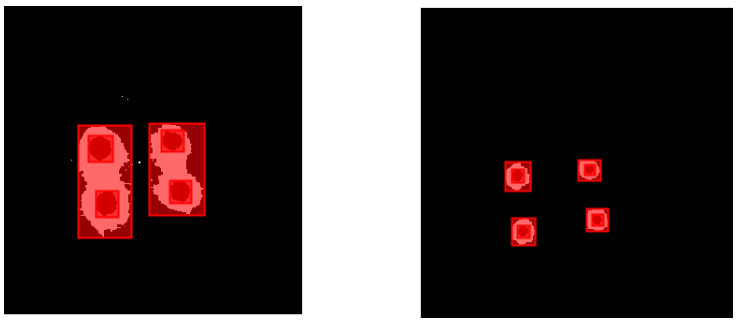

Figure 2.9: Antenna Recognition boxes. The old method is on the left, and the new method is ont the right

filter values in the manner shown in Figure 2.8. The two elongated halos surround two LEDs each were reduced to four smaller halos each surrounding a single LED. This required an adjustment to how the boxes drawn around sharp contours were interpreted. Figure 2.9 illustrates the change. The old location algortithm looked for 6 rectangles in total, with two smaller rectangles inside each of the two larger rectangles. The new algorithm is modified to look for eight rectangles. Each of the four larger rectangles should contain a single smaller rectangle inside it. Furthermore, due to the separation of the LEDs around the antenna, the antenna can now be rotated without breaking the algorithm as the larger rectangles no longer overlap. A successful antenna location during rotation is shown in Figure 2.10. This modification of the filter value had the additional benefit of being less sensitive to ambient light, which increased the location success rate.

2.2.2 Different LED Configurations

One intriguing extension to the computer vision system in its current state is the ability to distinguish between different types of objects based on differing LED configurations. To demonstrate this, the LED controllers were modified so that the fourth LED could be toggled on and off, and the detection system was modified to allow for three large rectangles containing one small rectangle

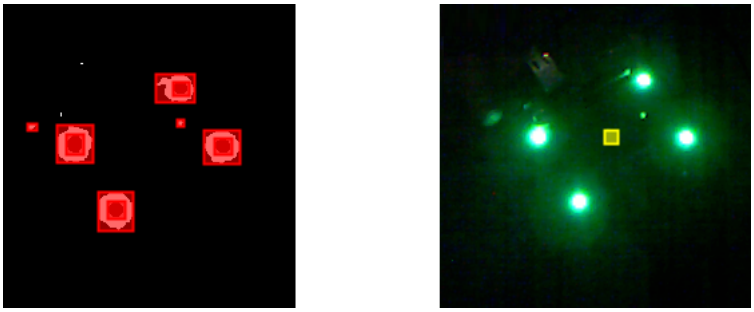

Figure 2.10: Antenna Recognition with a Rotation

each to count as a recognition case.

2.2.3 Machine Learning Techniques

With all these modifications, the computer vision system still has the rather serious limitation of only detecting one antenna at a time. With the existing approach using boxes to detect antennas, a potential algorithm to provide for a successful antenna detection becomes more exponentially more complex as the number of antennas increases. This situation is therefore conducive to alternate approaches, particularly machine learning approaches.

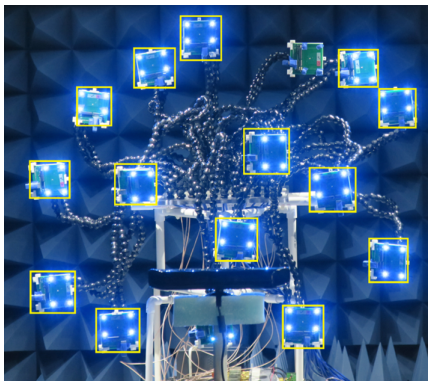

Figure 2.11: Detecting multiple antennas at once

Machine learning already finds widespread use in facial recognition software, and that application has enough similarities to detecting different antennas within the picture to make it feasible to

leverage the work already done. The approach involves 'training' the machine learning algorithm with a set of images of a single antenna at various angles and various levels of ambient lighting, then using the algorithm in an attempt to detect individual antennas from a large group like in Figure 2.11.

Work on this is started by filming a thirty second video of a single antenna and moving the camera around through a variety of angles. The video is split into all of its frames, and each frame is fed into the facial recognition software as a training set. The open source software is then tested with an antenna array to look for results.

3. RESULTS

The methods for developing and testing random antenna arrays focused on in this paper often do not have a quantitative measure for their effectiveness. Where one can be approximated, that approximation is described; otherwise, qualitative measures of success are used.

3.1 Electromagnetic Simulations

The HFSS Automation framework was successful in significantly reducing the amount of time it takes to model a large random array in the software. Table 3.1 shows some examples of approximated model construction times by hand compared with measured times for the scripted modeling. Manual construction is done using a template file. The scripted modeling time does not scale linearly, because HFSS responds to commands slower as the model size increases. The

Model	Construction Method	Time(hh:mm:ss)
Single Patch Antenna	Designed by Hand	00:30:00
Single Patch antenna	Python Script	00:00:05
16 Element Random Antenna Array	Designed by Hand	01:00:00
16 Element Random Antenna Array	Python Script	00:01:19
32 Element Random Antenna Array	Designed by Hand	01:45:00
32 Element Random Antenna Array	Python Script	00:05:06

Table 3.1: Time reduction automating HFSS vs CADing by hand

dual quaternion interface works, and the framework is able to model systems designed using dual quaternion coordinates

3.1.1 Known Issues

When phasing is applied to these models, the resulting waveform does not appear to be as directive as some models completed by hand. This is likely due to to the coordinate systems in the script having an origin in a location slightly offset from the feed point, which should be fairly straightforward to correct.

3.1.2 Continuing Work

Additional features can be added to the library in its current state. One example would be the ability to take a template file of a unique antenna, and quickly replicate it into an random (or ordered) array. Other potential areas for improvement include developing an interface for complex fractal or parametric structures.

3.2 Computer Vision System

The major success with the computer vision system was the ability to take measurements and successfully demonstrate beamforming with rotated antennas. This fix allowed for other projects using Medusa in the lab to progress beyond this roadblock.

The machine learning approach to antenna recognition, while promising, is still in the early stages of implementation, and therefore have no significant results to show for it yet. This is anticipated to be a continuing area of exploration in the coming months.

4. CONCLUSIONS

This paper presented improvements to the current approach to simulation and testing of random antenna arrays. The automation framework for HFSS has been successfully implemented, and with a few bug fixes should see use in the lab setting immediately. The fixes to the computer vision system enabled other projects to move forward and take data with rotated antennas which previously could not be measured. The approach to antenna recognition with machine learning techniques similar to facial recognition is in progress, and shows promise for results in the near future. Through further work on this system, the Huff Research Group and other researchers utilizing random antenna arrays will be able to quickly measure and simulate test setups thereby making further advancements in the antenna community more accessible.

REFERENCES

[1] K. Buchanan and G. H. Huff, "A Stochastic Mathematical Framework for the Analysis of Spherically-Bound Random Arrays," *IEEE Transactions on Antennas and Propagation*, vol. 62, pp. 3002–3011, June 2014.

[2] Y. Lo, "A mathematical theory of antenna arrays with randomly spaced elements," *IEEE Transactions on Antennas and Propagation*, vol. 12, pp. 257–268, May 1964.

[3] Y. Lo, "Sidelobe level in nonuniformly spaced antenna arrays," *IEEE Transactions on Antennas and Propagation*, vol. 11, pp. 511–512, July 1963.

[4] K. Buchanan, J. Rockway, and G. H. Huff, "Random antenna array phase and range limitations," in *2015 IEEE International Symposium on Antennas and Propagation USNC/URSI National Radio Science Meeting*, pp. 2499–2500, July 2015.

[5] K. R. Buchanan, *Theory and Applications of Aperiodic (Random) Phased Arrays*. Thesis, May 2014.

[6] L. M. Aristizabal and C. A. Zuluaga, "Distance measure with computer vision and neural networks for underwater applications," in *2016 IEEE Colombian Conference on Robotics and Automation (CCRA)*, pp. 1–6, Sept. 2016.

[7] J. S. Goddard and M. A. Abidi, "Pose and motion estimation using dual quaternion-based extended Kalman filtering," vol. 3313, pp. 189–200, 1998.

[8] S. R. Chapala, G. S. Pirati, and U. R. Nelakuditi, "Determination of coordinate transformations in UAVS," in *2016 Second International Conference on Cognitive Computing and Information Processing (CCIP)*, pp. 1–5, Aug. 2016.

[9] J. McDonald, "Teaching Quaternions is not Complex," *Computer Graphics Forum*, vol. 29, pp. 2447–2455, Dec. 2010.

[10] K. Joy, “Geometric Modeling Lectures – Quaternion Transformations.”

[11] M. Abrate, “Qmath Quaternion Algebra Package,” Mar. 2012.

[12] J. Ruff, “Huff Research Group Github Repository – HFSS Python Automation Framework.”

APPENDIX A: APPLICATION LAYER SOURCE CODE

The code used for the HFSS automation framework is stored in full on the Huff Research Group Github page[12]. The repository contains the following files:

- $HFSSLibrary.py$ contains all the application layer functions discussed in section 2.1.2.
- $EmagDevices.py$ is the full source for the device layer discussed in 2.1.3.
- $main.py$ is the full example main file shown in part in section 2.1.4.
- $qmathcore.py$ is the quaternion math class converted from the python 2 source taken from[11].
- $DualQuaternion.py$ the dual quaternion math class converted from the python 2 source taken from[11].