Optimization of Transverse Flux Motor for Utilization in Bionic Joints

by

Cameron Roy Taylor
B.Sc., Brigham Young University, 2014

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
in Partial Fulfillment of the Requirements for the Degree of

Masters of Science in Media Arts and Sciences
at the
Massachusetts Institute of Technology

June 2016

© 2016 Massachusetts Institute of Technology. All rights reserved.

Signature redacted
Signature of Author

Program in Media Arts and Sciences
May 6, 2016

Signature redacted
Certified by

Hugh Herr
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Signature redacted
Accepted by

Pattie Maes
Academic Head, Program in Media Arts and Sciences
DISCLAIMER NOTICE

Due to the condition of the original material, there are unavoidable flaws in this reproduction. We have made every effort possible to provide you with the best copy available.

Thank you.

Figure 4.1 is missing/ommitted from p.58
Optimization of Transverse Flux Motor for Utilization in Bionic Joints

by

Cameron Roy Taylor
B.Sc., Brigham Young University, 2014

Submitted to the Program in Media Arts and Sciences,
School of Architecture and Planning,
on May 6, 2016
in Partial Fulfillment of the Requirements for the Degree of

Masters of Science in Media Arts and Sciences
at the
Massachusetts Institute of Technology

ABSTRACT

Though there have been remarkable advances in powered prosthesis technology over the past decade, design limitations of commercial electric motors are one of the main bottlenecks in meeting critical device requirements, such as minimum range on a single battery charge and acoustic emission restrictions. Traditional motor design focuses on motor development for operation at specific torques and velocities, but a motor design which minimizes the power loss over the torque-velocity profile of a bionic ankle is more precisely what is needed for our application.

Considering the design requirement in this way lays the groundwork for a new design framework. Leveraging this problem statement, we herein develop a new motor design process generalizable to all applications requiring a variable but cyclic torque-velocity profile. We present a motor optimization package for cyclic variable torque-velocity motor design and demonstrate its viability in constrained optimization of a transverse flux motor for use in a bionic ankle. We further evaluate and present the intended use of this transverse flux motor for application in bionic joints, along with advantages and design hurdles of the planned system.

Thesis Supervisor: Hugh Herr
Title: Associate Professor of Media Arts and Sciences
Optimization of Transverse Flux Motors for Utilization in Bionic Joints

by

Cameron Roy Taylor
B.Sc., Brigham Young University, 2014

Signature redacted

Advisor ____________________________
Dr. Hugh Herr
Associate Professor of Media Arts and Sciences
MIT Media Lab

Signature redacted

Reader ______________
Dr. James L. Kirtley, Jr.
Professor of Electrical Engineering
MIT

Signature redacted

Reader ______________
Dr. Joseph A. Paradiso
Professor of Media Arts and Sciences
MIT Media Lab
# Contents

Abstract 3

Table of Contents 7

List of Figures 9

List of Tables 11

Acknowledgements 12

1 Introduction 13
   1.1 The Need for Improved Prosthesis Technology 13
   1.2 Impact of Prosthesis Design Advancements on Other Populations 15
   1.3 Outline 15

2 Novel Optimization Technology for Variable Torque-Speed Applications 17
   2.1 The Need for a New Design Framework 17
   2.2 Using the Torque-Velocity Profile to Design an Application-Specific Motor 17
   2.3 Calculation of Loss Constants 18
   2.4 Calculation of Average Power Loss Over One Cycle 22
   2.5 Content, Structure and Operation of Motor Optimization Package 24
   2.6 Software Integration Between Python, JMAG, and SolidWorks 25
   2.7 Necessity of a Log File for Records and Restarts 26
   2.8 Linear Mapping of Optimization Cost Function Inputs 27

3 Application of the Novel Optimization Routine to a Fundamental Configuration of a Novel Transverse Flux Motor 31
   3.1 Structure and Principles of Operation 31
List of Figures

Figure 1.1 BiOM on Patient ................................................. 14
Figure 2.1 Motor Torque-Velocity Profile: Gait Data from a Powered Prosthesis 18
Figure 2.2 Motor Torque-Velocity Profile: Torque and Velocity Plotted Separately 19
Figure 2.3 Single-Case Fits of the Various Constants for Loss and Torque . . . 21
Figure 2.4 Logarithmic Torque Relationship Fit .................................. 22
Figure 2.5 Adapted Motor Torque-Velocity Profile for our New Implementation 24
Figure 2.6 Motor Optimization Flowchart ...................................... 29
Figure 2.7 Motor Optimization Function Tree .................................. 30
Figure 3.1 Single-Phase Transverse Flux Motor ................................. 32
Figure 3.2 Single-Phase Pi-Section with Magnetic Field Vectors ............. 33
Figure 3.3 Three-Phase Transverse Flux Motor .................................. 35
Figure 3.4 Biom Cutaway ..................................................... 36
Figure 3.5 BiOM Open ......................................................... 37
Figure 3.6 Integration of Motor and Ball Screw ................................. 38
Figure 3.7 Next-Generation Ankle Design .................................... 39
Figure 3.8 Next-Generation Bionic Ankle Angles Diagram ................... 40
Figure 3.9 Transmission Ratio Dependence on Ankle Angle .................. 41
Figure 3.10 Single-Phase Pi-Section with Labeled Optimization Parameters . 42
Figure 3.11 Comparison Between Torque in Three Phases .................... 44
Figure 3.12 Three-Phase Transverse Flux Motor Pi-Section .................. 45
Figure 3.13 Pareto Frontier for Various Mesh Element Sizes .................. 46
Figure 3.14 Torque Over Time Given Fine and Rough Step Sizes and Error at
   Shared Steps ................................................................. 46
Figure 3.15 Pole Iron Loss Approximation as a Function of Pole Radial Extent
   and Pole Width ............................................................. 48
Figure 3.16 Second Half Simulated and Predicted Voltage, Torque, and Loss . 50
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Five Parameter Optimization Line Plots</td>
<td>54</td>
</tr>
<tr>
<td>4.2</td>
<td>Five Parameter Optimization Andrew’s Curves</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Five Parameter Optimization Scatter Plots</td>
<td>56</td>
</tr>
<tr>
<td>4.4</td>
<td>Mass Vs. Power Loss for the Transverse Flux Motor Designs</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Maxon EC30 Powermax</td>
<td>59</td>
</tr>
</tbody>
</table>
List of Tables

Table 4.1  Comparison of Motor Implementations in the Bionic Ankle  . . . . . .  58
ACKNOWLEDGEMENTS

I must first thank my advisor, Hugh Herr, for giving me the opportunity to work on this motor optimization project and for granting me the freedom and trust to experiment with many possibilities in developing this electric motor design process. This freedom ultimately enabled my success in this project. A special thanks is also due to my readers, James Kirtley and Joseph Paradiso. Your insightful questions and your guidance had an appreciable impact on this work.

To David Calley and all of his team at Planet Rider, LLC., I express my appreciation for your collaboration in this project, for your inspiring creativity, and for the many opportunities to solve problems from the seat of a bicycle.

I must also acknowledge the contributions of Chris Williams, for your mechanical design experience, Ken Pasch, for your electromagnetic design experience, and Rick Casler, for your actuator design experience. Additionally, to Bruce Deffenbaugh, I owe gratitude for your suggestions in implementation of the optimization algorithm. To all of you, your insights formed the foundation for this project, and I express my sincere gratitude for your guidance and direction.

To all the members of the Biomechatronics Group who have listened, brainstormed with me, mentored me, encouraged me and inspired me, especially Luke Mooney, Michael Eilenberg, Madeleine Abromowitz, Jean-Francois Duval, Tyler Clites, Kevin Moerman, Arthur Petron, Sophia Tabchouri, Matthew Furtney, Roman Stolyarov, David Sengeh, Stephanie Ku, Christophe Everarts, David Hill, and Lindsey Reynolds, and to the members of the Media Lab who have extended their support in this project, including Joost Bonser, David Ramsay, Sam Calisch, Tom Lutz, and John DiFrancesco, thank you for your patience, time, ingenuity, compassion and friendship.

It is only appropriate for me to thank my parents for their consistent expressions of confidence in my success and in the success of my siblings. Without your assurances, I would not have had the courage to be where I am today.

Finally, to my wonderful wife, Sara, thank you for encouraging me in my constant quest for innovation, for being my teammate in engineering and in life, and for your words of rhyme and reason which keep me calm, collected, and consistent.
Chapter 1

Introduction

1.1 The Need for Improved Prosthesis Technology

Approximately 185,000 amputations are performed in the United States every year, and this number is expected to double by the year 2050 [18]. Among these, 65% are lower extremity amputations. Simultaneously, lower extremity prostheses are on a path to being superior to their biological counterparts [8]. With the development of the powered ankle prosthesis [1, 2, 3], the improvement of comfortable prosthetic sockets [14], and research invested into control of prostheses via brain-controlled interfaces [10], the perception of lower extremity amputation as a disability is gradually becoming a thing of the past.

The early design work for the first commercially available powered prosthesis was done at MIT [1, 3]. Careful focus was placed on the design trade-off between energy and power output and the size and weight of the device. Ultimately, the design that was chosen for production was similar in mass and volume to the biological counterpart that the device was replacing [2].

The PowerFoot BiOM powered ankle prosthesis (see Figure 1.1), currently the only prosthesis on the market capable of delivering powered plantarflexion (foot movement towards the sole), reduces the metabolic cost of walking for a user to approximately that of a non-amputee [4, 7]. Because of this, transtibial amputees are able to recover many of the social and physical benefits of walking. In addition, it has been validated that powered ankle prostheses can reduce painful secondary impairments caused by walking with prostheses, such as knee osteoarthritis [12].

Yet the characteristics of the biological leg still remain the gold standard for achievement in quiet, efficient operation. The average person walks about 3,000 gait cycles (6,000
Figure 1.1: The PowerFoot BiOM powered ankle prosthesis normalizes gait and restores function in lower extremity amputees. Image Credit: Business Wire

steps) each day [17], all without making a noise detectable to the human ear. According to information from the product website, “the BiOM T2 System can typically travel approximately 1,500-2,000 strides per battery depending on user speed, activity and weight” [6]. Thus, in order to walk the number of gait cycles required for baseline mobility and social benefits, users of the BiOM must carry with them two to three additional batteries in order to power the ankle through the day. Though significant work is being done to improve the specific energy of portable power sources, we have yet to see a radical development become commercially viable for application in this field. Further, the motor controller switches off when the battery exceeds “150°F while walking”, which typically occurs at about 600 steps [5]. In order to increase both the range on a battery charge and this thermal limit, a more efficient actuator (motor and transmission) is required.

This powered prosthesis is equipped with a Maxon brushless DC motor which is required to run at high-speed in order to achieve maximum efficiency for the motor. Unfortunately, this high required speed produces unfavorable acoustic emissions. The social stigma around the sound of the actuator, which is similar to that of squeaky shoes, causes some transtibial amputees to decide against using a powered prosthesis. Over 70% of the acoustic emissions from the BiOM at walking speeds over 1.25 m/s are caused by the belt-drive portion of the transmission. The first harmonic from the belt lies in the 1.5-3 kHz range, where human hearing is especially sensitive. A motor designed for high torque operation at lower speed would allow the removal of the belt-drive, and thus reduce these acoustic emissions.
1.2 Impact of Prosthesis Design Advancements on Other Populations

The solutions to the present-day shortcomings in prosthesis design will immediately have an impact upon other populations such as strokes victims and patients with spinal cord injuries. A growing elderly population amplifies the need for assistive mobility robotics, which must be improved both acoustically and in efficiency before being distributed to the population at large.

More boldly, the author of this thesis suggests that the principles developed and explained herein are the first steps toward an incredibly diverse set of robotic actuators, ultimately altering the way that we perceive and interact with the machines around us.

1.3 Outline

In Chapter 2, we discuss a new design framework for the design of electric motors in cyclic variable torque-velocity applications. In Chapter 3, an example application is presented, and the implementation of this new design framework for a novel motor topology is described. The results of this design process are presented and discussed in Chapter 4. Chapter 5 discusses the future work in developing the optimization software, as well as the impact of this work on a more general class of machines.
Chapter 2

Novel Optimization Technology for Variable Torque-Speed Applications

2.1 The Need for a New Design Framework

The traditional model of selecting a motor design involves choosing a motor that can efficiently perform at a discrete number of torques and speeds. Due to the emerging need for motors which can efficiently handle large variability in torque and speed, it is vital that we develop a new way of thinking about and designing electric motors. To meet the efficiency and acoustic constraints of the bionic ankle, we must take into account the continuous torque-velocity profile\(^1\) of the human gait cycle in designing the optimal motor. This design framework is described below.

2.2 Using the Torque-Velocity Profile to Design an Application-Specific Motor

Although a variety of new applications require high variability in torque and velocity over time, many modern machines are also cyclic in nature (see Figures 2.1 and 2.2 for an example of our specific application). This allows us to focus on one machine cycle in our new design framework. We have created a novel optimization framework that takes the torque-velocity profile of an application as an input and outputs the average power loss

\(^1\)In this thesis, the “torque-velocity profile” refers to the continuous curves of torques and speeds required over a single actuator cycle. This is different from the torque-speed curve, which defines the maximum operating limits of an electric motor.
Figure 2.1: Motor Torque-Velocity Profile: Gait Data from a Powered Prosthesis. Color represents the time from heel-strike, with the gait cycle from heel-strike to heel-strike taking 1.05 seconds. During the yellow-green portion of the curve, the motor charges up a spring, which is released in the green-cyan portion of the curve during plantarflexion.

over one cycle as the output of the cost function.

Our optimization package characterizes the power loss of the motor as a function of torque and velocity using magnetic modeling software and uses the torque-velocity profile required by the load to calculate the average total power losses (iron loss and copper loss) over one machine cycle. An explanation of the operation of the package follows.

2.3 Calculation of Loss Constants

In order to characterize a motor model in such a way as to calculate the average power loss over a gait cycle, one must compile a list of equations that describe the losses as a function of motor behavior. It is well known that there is a relationship between squared torque and copper, or resistive, loss, and that there is an exponential relationship between frequency
Figure 2.2: Motor Torque-Velocity Profile: Torque and Velocity Plotted Separately. Here, the torque and velocity requirements shown in Figure 2.1 are shown on separate plots. The color labelling, though unnecessary for the figure, is preserved for convenience in comparing with Figure 2.1.

and iron loss. In order to determine these relationships, it is necessary to simulate the operation of the motor at several torques and velocities. It was chosen to use three torque-velocity cases at which to simulate the power loss. These three cases were low current and low velocity, medium current and medium velocity, and high current and high velocity. Low, medium, and high values were determined by experimentation, choosing a velocity at the top of the required range for our application, and choosing a current which would deliver approximately the maximum amount of torque needed for our application. The low and medium torques and velocities were then spaced evenly from zero to the high case.

It is a common assumption that copper loss is dependent only on torque and that iron loss is dependent only on frequency. These assumptions are leveraged in the optimization, and the low, medium, and high cases are simulated with velocity and torque increasing together, so that three cases are tested instead of nine, reducing the simulation time by
two-thirds.

The squared torque copper loss proportionality constant, \(k_{\text{copper}}\), (the relationship between average copper loss and squared average torque) was determined by a proportional fit, so that the relationship

\[
p_{\text{copper}} = k_{\text{copper}} T_{\text{avg}}^2
\]  

(2.1)

can be used in calculation of average copper loss. This constant was calculated with a linear least squares fit, solving

\[
\begin{bmatrix}
T_{\text{avg}}^2 \\
T_{\text{avg}}^2 \\
T_{\text{avg}}^2
\end{bmatrix}
\begin{bmatrix}
k_{\text{copper}}
\end{bmatrix}
=
\begin{bmatrix}
p_{\text{copper avg}1} \\
p_{\text{copper avg}2} \\
p_{\text{copper avg}3}
\end{bmatrix}
\]  

(2.2)

where \(T_{\text{avg}i}^2\) is the square of the average total torque for the \(i^{th}\) case, and \(p_{\text{copper avg}i}\) is the average total copper loss corresponding to the same case. Solving for \(k_{\text{copper}}\) gives

\[
k_{\text{copper}} = \frac{T_{\text{avg}1}^2 p_{\text{copper avg}1} + T_{\text{avg}2}^2 p_{\text{copper avg}2} + T_{\text{avg}3}^2 p_{\text{copper avg}3}}{T_{\text{avg}1}^4 + T_{\text{avg}2}^4 + T_{\text{avg}3}^4}
\]  

(2.3)

and simplifying we get

\[
k_{\text{copper}} = \frac{T_{\text{avg}1}^2 p_{\text{copper avg}1} + T_{\text{avg}2}^2 p_{\text{copper avg}2} + T_{\text{avg}3}^2 p_{\text{copper avg}3}}{T_{\text{avg}1}^4 + T_{\text{avg}2}^4 + T_{\text{avg}3}^4}
\]  

(2.4)

In practice, this was implemented in the form of Equation 2.3 to maintain generalizability of the vector equation should more points along the characteristic curve be used in the characterization of the motor in future use of the package. See Figure 2.3(b) for an example of this fit.

Again using traditional motor design assumptions, the relationship between iron loss and frequency was assumed to be

\[
p_{\text{iron}} = k_{\text{iron}} f_{\text{iron}}
\]  

(2.5)

As this cannot be implemented in a straightforward manner with a matrix equation, these parameters were solved by minimizing the least squares error. See Figure 2.3(a) for an example of this fit.
Figure 2.3: Single-Case Fits of the Various Constants for Loss and Torque. In Subfigure (a), secondary iron loss, consisting of the iron loss from magnets, flux concentrators, and the flux return, is fit with a curve of type $k_f f^a$. Subfigure (b) shows the copper loss (the loss in the coils), being fit with a curve of type $k_f^2$. Subfigure (c) shows the torque constant being fit with a curve of type $\frac{k}{a} \ln(aI + 1)$, which is $k_I$ at small values of $I$, but falls off as $I$ increases. Subfigures (a) and (b) are used to compute losses, while Subfigure (c) is used to compute maximum required voltage amplitude.

Finally, to determine the maximum required current output, it was necessary to determine a relationship between torque and current. The traditional assumption that torque is proportional to current,

$$\tau_{avg} = k_I I$$

(2.6)

was initially used, and the the torque constant $k_I$ was solved for using the same method as was used to solve Equation 2.3. However, the torque was found to saturate within the operation range, so it was determined that a new equation be used to model the relationship between torque and current. The curve fit

$$\tau_{avg} = k_I \frac{\ln(a_{kt} I + 1)}{a_{kt}}$$

(2.7)

was used instead, because it has a derivative with respect to $I$ of $k_I$ when $I = 0$, but the slope falls toward zero as $I$ increases, as

$$\frac{d\tau_{avg}}{dI} = \frac{k_I}{a_{kt} I + 1}$$

(2.8)

To visualize the impact of various values of $a_{kt}$ from Equation 2.7 on the shape of the torque-current curve, see Figure 2.4. Note that an increase in $a_{kt}$ corresponds to the non-
linear region beginning at a lower electric current value.

Figure 2.4: Logarithmic Torque Relationship Fit. This plot of Equations 2.6 and 2.7 shows the impact of various values of $a_{kt}$ when $k_i$ is fixed to 1. Note that an increase in $a_{kt}$ corresponds to a fall-off in torque at a lower electric current value, but that each realization of Equation 2.7 has a corresponding linear region.

After calculating the maximum torque needed in the motor, the inverse of Equation 2.7 was used to calculate the maximum current required, as

$$I_{max} = e^{\frac{a_{kt}t_{max}}{k_i}} - 1$$  \hspace{1cm} (2.9)

Since a three-phase motor is controlled with a wye-connection, the maximum current amplitude required by the controller will never be higher than the maximum current amplitude required by a single phase of the motor, so this maximum current represents the maximum current required by the controller as well.

### 2.4 Calculation of Average Power Loss Over One Cycle

Once each motor configuration was characterized using the aforementioned three cases in simulation, Equations 2.1 and 2.5 were used to calculate the total power loss of the motor given the torques and velocities of a single gait cycle. In order to do this, the electrical
frequency was calculated from each of the three speeds using the pole count, and the torque values corresponding to each of the three currents in simulations were stored and used to compute the average total torque. Iron loss and copper loss values were then imported from the simulation data and stored as averages.

Next, the required torque-velocity profile was input into the optimization. Using the geometry and material parameters of a particular motor configuration, the rotor inertia was calculated. The velocity was then differentiated using a standard five-point stencil, as well as four five-point boundary stencils used to reduce calculation error at the boundaries (see Equation 8 of Lakin 1986 [11] for the finite difference stencil used and Computational Science and Engineering [15] for examples of forward and backward differences in common use). The rotor inertia of the previous motor was removed from the torque-velocity profile of our application, and the calculated rotor inertia of each motor configuration was incorporated into the torque-velocity profile using the required motor acceleration. See Figure 2.5 for an example of this transformed torque velocity curve. Note that in addition to the the transformation for motor inertia, that there is a difference in scaling. Section 3.3 discusses the change of transmission ratio which caused this scaling.

Using the loss constants and this transformed torque-velocity curve, the power loss was computed for each torque-velocity value, and then the average over all of these power losses was computed. This is equivalent to computing the Riemann sum to calculate the total energy loss over all data points in the torque speed curve

\[
E_{loss} = \sum_{i=1}^{N} \left( p_{copper}(\tau_i, \omega_i) + p_{iron}(\tau_i, \omega_i) \right) \Delta t
\]

\[
= \Delta t \sum_{i=1}^{N} p_{copper}(\tau_i, \omega_i) + p_{iron}(\tau_i, \omega_i)
\]

(2.10)

and dividing by the gait cycle period, \( N \Delta t \), where \( N \) is the number of data points and \( \Delta t \) is the sampling interval, to get

\[
P_{loss} = \frac{1}{N} \sum_{i=1}^{N} p_{copper}(\tau_i, \omega_i) + p_{iron}(\tau_i, \omega_i)
\]

(2.11)

since \( P_{loss} = \frac{E_{loss}}{N \Delta t} \). With this observation, average power loss over the gait cycle was computed as an average of the power losses computed at each torque and speed in the torque speed profile, with total energy loss then being computed as \( E_{loss} = N \Delta t P_{loss} \).

This average power loss was then returned to the optimization function as the output of
Figure 2.5: Adapted Motor Torque-Velocity Profile for our New Implementation. Note the similarity to Figure 2.1. This new torque-velocity profile was obtained by reducing the transmission ratio of the actuator by a factor of 11 and accounting for the increase in rotor mass from the Maxon EC30 to the transverse flux motor.

2.5 Content, Structure and Operation of Motor Optimization Package

We created our optimization package in Python. The optimization as well as pre-simulation and post-simulation calculations are performed in Python functions, and the cost function includes a driver for Python. In order to minimize the size of the driver, a model is fully set up in the magnetic modeling software, and the driver duplicates the model, altering only the parameters and conditions pertaining to the particular optimization. This was found to reduce setup errors in the model. When the input involves geometric parameters, the
driver commands the magnetic modeling software to alter these parameters in SolidWorks (Dassault Systèmes) through a CAD link. See Figure 2.6 for a diagram of the optimization package.

In order for the cost function to determine when to request results from the magnetic modeling software, it is critical to monitor progress. If this is implemented incorrectly, results are requested early and cause simulations to halt, or results are never requested. Additionally, any error in the code may cause the progression to the next step of the optimization to fail, requiring a manual restart. The function for monitoring progress of the simulations checks two script files used by the magnetic modeling software, and through empirical understanding of the contents of the scripts and the number of files generated by the simulation, the progress of the optimization is able to be determined with precision greater than that which is displayed by the magnetic modeling simulation progress interface.

An interface for visualizing the progress of the data is included in the motor optimization package, and the Python Pandas package is used for all data management. Use of careful data management was critical for processing the data, as the order of the data extracted from the magnetic modeling software was found to be inconsistent, but was clearly labeled, making processing especially straightforward using Pandas. Additionally, the motor optimization package was separated into several different functions for simplicity in debugging and for modularity. See Figure 2.7 for the hierarchy of these functions in the optimization process.

The Nelder-Mead, or downhill simplex, method was chosen for the first use of this package because it is a simple optimization method. However, this package can be utilized using any optimization algorithm. If the optimization is one of a random nature, then a fixed seed will be required for use of the power failure restart feature.

2.6 Software Integration Between Python, JMAG, and SolidWorks

The magnetic modeling software used in our application is JMAG, a commercial software from JSOL Corporation, distributed in North America by Powersys Solutions.

There were three code language options for driving JMAG: VBScript, JScript, and Python. Python was chosen because of the many packages available for data manipulation and optimization. WinPython was used, with Spyder as the IDE, for early work. It was
found necessary to run the code from the command line to run the file continuously for several weeks at a time.

In order to manipulate the motor geometry in the SolidWorks file via the JMAG CAD link, it was necessary to structure the file so that the geometry of each of the parts comprising the assembly could be accessed. There are several different methods for doing top-down design, and none stand out as the best solution. Four methods were considered and experimented upon: (1) altering of a text file referenced by all parts, (2) linking SolidWorks to Excel to create a design table, (3) creation of an assembly from a part parent structure via multi-body parts or parent sketches, and (4) creation of virtual parts in an assembly from Layout Blocks. The first two methods were found to be viable, but required careful attention to linking and rebuilding. The third method did not grant access via JMAG or python to the geometry without the SolidWorks API. For simplicity, this was not used. The fourth method, creating virtual parts in an assembly, allowed JMAG access of all of the variables in SolidWorks. It was discovered by experience that there is a set of nuances to be cautious of in the interfaces both from the JMAG perspective and from the SolidWorks perspective (i.e. the particular importance of saving a SolidWorks assembly before creation of blocks and parts in order to preserve these links and the need for particular care in using the Pack & Go function to duplicate SolidWorks assemblies), but the links between an assembly and virtual parts using the fourth method allowed for the entire SolidWorks assembly to be updated simultaneously by JMAG upon command. For this reason, the virtual parts method was chosen for implementing the optimization.

2.7 Necessity of a Log File for Records and Restarts

All pertinent outputs for each of the simulated cases are stored for comparison against imposed constraints in the optimization. When the optimization requests evaluation of an input which is outside the boundaries of the optimization, the algorithm immediately returns a high value without performing the simulation. Otherwise, the configuration is characterized and the characterization is evaluated against the constraints before returning the average power loss to the optimization function.

This storage is also vital for tracking the progress of the optimization, viewing the optimization history, and for recovering from power surges and power outages. As optimization algorithms do not typically create a record of each tested point, it was important to save the history for feeding stored results back to the optimization algorithm should a computer failure cause the optimization to stop short of convergence. The average power loss is thus
stored in a log file along with the variables for the corresponding configuration, to be used as a look-up table in case of restart.

The results are also fully stored in CSV (comma-separated value) files, and the functions created for reading and processing the average power loss can be re-applied for viewing the entirety of the results post-optimization. Thus, loss or deletion of the original results files from within the magnetic modeling software does not result in the loss of the results. This is an especially important step, since the raw results files for a magnetic modeling simulation can require memory storage on the order of gigabytes per simulation, while a csv file of pertinent results requires storage on the order of kilobytes.

### 2.8 Linear Mapping of Optimization Cost Function Inputs

The downhill simplex method of optimization works by creating a simplex (the generalization of a triangle) in n-space, and reflecting the worst vertex of the simplex about the centroid of all other points in the simplex. This is accompanied by expansions, contractions, and reductions in the simplex as it moves through the input space.

It was discovered that the particular implementation of the downhill simplex method in the python scipy package used creates the initial simplex by scaling each of the initial inputs up individually by 5%. Originally it was determined that the value halfway between the bounds of each input should be used for the initial input vector. However, it was found that this constant scaling ratio was creating a simplex skewed in favor of larger variation, and thus greater exploration by the optimization method, over the space of the inputs with the largest initial values. This issue occurs whenever input values, whether geometry or otherwise, are of greatly different size (i.e. optimization of a coil radius vs optimization of a magnet thickness). In order to correct for this, it was determined that the inputs should be mapped to the motor geometry from a space giving equal variation opportunity to each of the input variables.

It was selected to still start all initial variables halfway between their respective bounds, but to also have an initial deviation from the value halfway between the bounds up by 25% of the difference between bounds. Expressed mathematically, this desired relationship is

\[ x_{\text{mid}} = 0.5(x_{\text{max}} + x_{\text{min}}) \]  \hspace{1cm} (2.12)

and
\[ x_{dev} = x_{mid} + 0.25(x_{max} - x_{min}) \]  \hspace{1cm} (2.13)

where \( x \) is a value in the input space. We know that we will start at \( x_{mid} \) and that

\[ x_{dev} = 1.05x_{mid} \]  \hspace{1cm} (2.14)

We wish to define this input space by its bounds, so we substitute Equation 2.12 into Equation 2.14, and substitute this result and Equation 2.12 into equation 2.13, producing

\[ 1.05 \cdot 0.5(x_{max} + x_{min}) = 0.5(x_{max} + x_{min}) + 0.25(x_{max} - x_{min}) \]  \hspace{1cm} (2.15)

which gives \( x_{max} = \frac{11}{9} x_{min} \). Defining \( x_{min} = 1 \) gives a fourth equation with which to determine our four variables, and we have \( x_{max} = \frac{11}{9} \) and \( x_{mid} = \frac{10}{9} \). We then map this space onto the space between the bounds of our motor parameters using the relationship

\[ y - x_{min} = \frac{y_{max} - y_{min}}{x_{max} - x_{min}} (x - x_{min}) \]  \hspace{1cm} (2.16)

where \( y \) is a value in the motor parameter space.

Making the entire input vector equal to \( x_{mid} \) starts the value of each motor parameter at the midpoint between its respective bounds, and this mapping gives a weighting to its variation proportional to the size of the difference between its bounds. This is important because it allows the optimization algorithm to more quickly transverse the breadth of the input space.
Figure 2.6: Motor Optimization Flowchart. The torque-velocity profile, shown in red, is input to the simulation, along with initial constants and a motor simulation model from which to create successive configurations. These motor configurations are created, simulated, and characterized in order to compute the performance of the motor over the torque-velocity profile, which is output to the optimization as average power loss. A log file is kept for records and restarts.
Figure 2.7: Motor Optimization Function Tree. The motor optimization package is divided into several different functions for simplicity and modularity. The hue represents the chronological involvement of the function in each iteration of the optimization, while color intensity represents depth of the function call.
Chapter 3

Application of the Novel Optimization Routine to a Fundamental Configuration of a Novel Transverse Flux Motor

There currently do not exist off-the-shelf electric motors within the size, power and efficiency requirements for a bionic ankle design meeting our design requirements. An improvement over conventional designs is needed. The transverse flux motor has been discussed as a motor design that leads to both higher power density and higher efficiency [9]. The transverse flux motor has been researched for and implemented in large-scale systems such as trains and elevators, but has not been researched or commercialized for small-scale applications. We have a novel transverse flux motor topology which we believe to be superior to alternative designs for application in bionic joints.

The Biomechatronics Group in the MIT Media Lab is collaborating with David Calley, owner of Planet Rider LLC, in miniaturizing and optimizing a novel transverse flux motor topology of his invention. Planet Rider has successfully built commercial transverse flux motors with high torque and high efficiency at low speeds. We have evidence in simulation suggesting that the transverse flux motor will be an efficient solution for driving the bionic ankle as well.

3.1 Structure and Principles of Operation

This novel transverse flux motor, shown in Figure 3.1, is composed of a rotor (shown in gray), and a stator. The rotor is composed of rectangular steel pleated into a flower shape,
where each of the petals constitutes a rotor pole. The two sides of the rotor are offset by 180 electrical degrees, and are connected by the flux return, a hollow cylinder also constructed of steel. The axis of this hollow cylinder is the axis of rotation for the rotor. The stator is composed of magnets (shown in blue) and magnetic material termed the flux concentrators (shown in red). A coil (shown in green) is positioned concentric to the magnet ring.

![Diagram of Single-Phase Transverse Flux Motor](image)

Figure 3.1: Single-Phase Transverse Flux Motor. Laminated steel sheets pleated into a flower shape create rotor poles (1), which are connected axially by a hollow cylindrical element called the flux return (2). Circumferentially-polarized magnets (3) alternate in polarization direction, pushing a spatially alternating magnetic field into the air gap through flux concentrators (4) of magnetic material. The coil (5) is located concentric to the ring of magnets and flux concentrators, termed the magnet ring.

See Figure 3.2 for the flux path of the motor. Magnets in the stator are polarized circumferentially and alternate in polarization direction. The flux concentrators between these magnets set up an axially-directed magnetic field which alternates spatially in the air gap. An alternating current in the coil drives magnetic flux radially inward through the far set of poles, through the flux return and back up through the near set of poles. The interaction between the temporally alternating magnetic field in the rotating poles and the spatially alternating magnetic field in the magnet ring creates a force in the circumferential direction.

As typical in electric motors, this single-phase design ideally creates torque in the form
Figure 3.2: Single-Phase Pi-Section with Magnetic Field Vectors. In this snapshot, the magnetic flux path can be seen as the magnetic flux flows through the magnets, into a shared flux concentrator, out the far side of the flux concentrator, and down the far rotor pole. Flux then returns through the flux return element (under the coil) and back up the near rotor poles, before crossing the air gap and entering the opposite flux concentrators. This flux is then directed back into the magnets. The high magnetic field intensity here is generated by electric current flowing through the coil. The interaction between the fields across the air gap caused the torque in the motor.
of a squared sinusoid. Thus, to get continuous torque, it is necessary to expand the design to multiple phases (we chose to expand the design to three), each offset by 120 degrees, since

$$\sum_{i=0}^{n} \sin^2 \left( \frac{2\pi i}{n} + t \right) = \frac{n}{2} \text{ for } n \in [3, \infty]$$  \hspace{1cm} (3.1)

where \(n\) is the number of phases (3, in our case), or by 60 degrees, since

$$\sum_{i=0}^{n} \sin^2 \left( \frac{\pi i}{n} + t \right) = \frac{n}{2} \text{ for } n \in [2, \infty]$$  \hspace{1cm} (3.2)

Traditionally, Equation 3.2 is used for spacing of the electrical cycle in a two-phase motor, but Equation 3.1 is used for three or more phases, since in practical application this allows the torque to be less impacted by shared asymmetries between the multiple phases, and because it allows for a wye-connected (or, more generally, star-connected) circuit configuration, though wire reversals can be employed for an odd number of phases to achieve this with Equation 3.2. We will follow the traditional torque summation method by using Equation 3.1.

The motor design we are working with involves a shifting by 180 electrical degrees of both the magnet ring and the electric current phase of the middle phase in addition to the 120 degree offset. This allows the inner phase to share rotor poles with the outer phases. The 180 electrical degree shift of the middle coil is implemented as a wire reversal so that a wye-connection can still be employed, but the offset in the magnet ring allows for torque generation following Equation 3.1.

This current reversal also carries importance in the flux sharing of rotor poles between phases. For a three-phase motor of this style without coil reversal in the middle phase, more flux is generated in the inner, shared poles than in the outer poles, since the currents are spaced at 120 electrical degrees, and \(\sin(t) - \sin(t + 120^\circ) = \sqrt{3}\sin(t - 30^\circ)\), suggesting a magnetic field intensity \(\sqrt{3}\) higher in the shared poles. Note that a subtraction is performed because the flux contribution to the rotor poles by the two currents is opposite, since Amperian loops around each of two adjacent wires are opposite in direction. This appears to have been noted by Washington, et. al.[16], as their transverse flux motor design incorporates a \(\sqrt{3}\) scaling in the shared poles to pass the additional flux without saturating the iron. With the inner coil reversed, however, this subtraction becomes \(\sin(t) - \sin(t + 300^\circ) = \sin(t + 60^\circ)\), indicating that the flux in the inner rotor poles is equal in amplitude to the flux in the outer rotor poles. Thus, this design makes it unneces-
sary to vary the axial dimension of the poles from section to section.

Additionally, note the structure of the laminations as shown in Figure 3.2. Laminations are created in the rotor by stacking several steel sheets before pleating the steel into the pole section configuration. These laminations are oriented in such a way as to be have magnetic flux cut tangentially to the laminations as much as possible during motor operation. See Section 3.8 for additional considerations in calculating the iron loss through the poles.

![Three-Phase Transverse Flux Motor](image)

Figure 3.3: Three-Phase Transverse Flux Motor. Motor volume, weight, and losses are reduced by sharing rotor poles between the phases. Magnet rings are offset mechanically by 120 electrical degrees, and a reversal of the middle coil allows the physical location of the poles to be offset for the middle phase by 180 electrical degrees.

### 3.2 Integration into the Bionic Ankle

The present-day BiOM powered ankle uses a DC motor attached to a timing belt to apply torque to a ball screw (see Figure 3.5). This creates a downward force, which is converted into a torque around the ankle joint through a torsion spring (see Figure 3.4).

The rotor consists of the poles and the flux return. In the planned design, the rotor is attached to a ball nut, which interacts with a concentric ball screw as in the case of a linear
Figure 3.4: PowerFoot BiOM Cutaway. The Maxon brushless DC motor (1) drives the ball-screw (3) through a timing belt (2). The ball-screw exerts a force on the carbon fiber series torsion spring (4), which provides a torque about the joint (5). *Image Credit: BiOM, Jimmy DeVarie*

stepper motor (see Figure 3.6). The lead of the ball screw is chosen to give the desired transmission ratio. The working end of the ball screw is still attached to the spring foot through a torsion spring. A reduced transmission ratio, along with a concentric ball screw configuration, removes the need for a timing belt. This results in a significant reduction in transmission acoustic output.

The planned implementation involves use of the BiOM structure with minimal modifications. As such, constraints on the radius and length of the motor are imposed by the shell of the bionic ankle.
Figure 3.5: BiOM Open. Shown from another perspective, the motor (1) is commanded by the control electronics (2) and interacts with the ball-screw (3) through the timing belt. The system is enclosed by an aesthetic and weather-resistant cover. Image Credit: Boston Magazine, Bruce Peterson

3.3 Transmission Ratio Selection and Calculation

The smallest available ball screw lead that could be chosen in an off-the-shelf ball screw meeting size constraints was selected. This was determined to be 12 mm. The transmission ratio was then calculated using this specification and a diagram of the next generation ankle (see Figure 3.8). Using $b$, the distance between ankle joint and motor mount, $r$, the fixed lever arm length, $l$, the ballscrew travel, and $\theta$ the reference angle, and with the law of cosines, we have
Figure 3.6: Integration of Motor And Ball Screw. In the planned implementation, the transverse flux motor, consisting of rotor poles (1), flux returns (2), magnets (3), flux concentrators (4) and coils (5) is contained within an aluminum case. Radial (8) and angular (9) bearings are used for motor rotation. The ball nut (7) rotates with the rotor, driving the ball screw (6) up or down accordingly. *Image Credit: Chris Williams*

\[ l^2 = b^2 + r^2 - 2br \cos(\theta + 90^\circ) \]  \hspace{1cm} (3.3)

simplifying, this becomes

\[ l^2 = b^2 + r^2 - 2br \sin(\theta) \]  \hspace{1cm} (3.4)

and solving for the reference angle yields

\[ \theta = \arcsin\left(\frac{l^2 - b^2 - r^2}{2br}\right) \]  \hspace{1cm} (3.5)

Differentiating gives
Figure 3.7: Next-Generation Ankle Design. In the planned implementation of the transverse flux motor in the bionic ankle, a concentric ball-screw motor design (1) as shown above in Figure 3.6 will be used. The battery (2) sits on the back of the ankle, as in the present-day bionic ankle. Space must be allotted for the motor control electronics (3). Image Credit: Chris Williams

\[ \frac{d\theta}{dl} = \frac{l}{br\sqrt{1 - \left(\frac{b^2 - r^2}{2br}\right)^2}} \]  \hspace{1cm} (3.6)

since \( \frac{d}{dx} \arcsin \left( f(x) \right) = \frac{f'(x)}{\sqrt{1 - f(x)^2}} \). Now using Equation 3.4 to solve this in terms of \( \theta \), we have

\[ \frac{d\theta}{dl} = \frac{\sqrt{b^2 + r^2 + 2br \sin(\theta)}}{br\sqrt{1 - \left(\frac{2br \sin(\theta)}{2br}\right)^2}} \]  \hspace{1cm} (3.7)

and simplifying again gives

\[ \frac{d\theta}{dl} = \frac{\sqrt{b^2 + r^2 + 2br \sin(\theta)}}{br \cos(\theta)} \]  \hspace{1cm} (3.8)

Noting that the transmission ratio is the relationship between a change in motor angle, \( \psi \),
Figure 3.8: Next-Generation Bionic Ankle Angles Diagram. Dimensions of the system including the motor, ball screw, and ankle at various angles were used to calculate the transmission ratio. Critical dimensions for this calculation were \( r \), the constant length of the lever arm, \( l \), the ball screw travel, and \( b \), the distance between the ankle joint and the motor sagittal rotation joint. The reference angle, shown in red, is defined as zero at perpendicular to the vector \( b \). Ankle angles, shown in blue, were calculated as an offset and a reversal post-calculation. Image Credit: Chris Williams

and a change in the reference angle, \( \theta \), we have a transmission ratio

\[
R(\theta) = \frac{d\psi}{d\theta} = \frac{d\psi}{dt} \frac{dt}{d\theta} \tag{3.9}
\]

where

\[
\frac{d\psi}{dt} = \frac{2\pi}{L} \tag{3.10}
\]

where \( L \), the ball screw lead, was chosen to be 12 mm, giving
Finally, negating the reference angle and offsetting it by an experimentally determined 14.2255° gives the ankle angle. The transmission ratio dependence on ankle angle is shown in Figure 3.9.

![Ankle Angle vs Transmission Ratio](image)

Figure 3.9: Transmission Ratio Dependence on Ankle Angle. The transmission ratio of the ankle is not a constant, but varies according to ankle angle from approximately 16 to 21. In the critical region, where plantarflexion occurs, the transmission ratio is approximately 20, which was the value used in the calculations. This curve was derived using the measurements in Figure 3.8.

Knowledge of this new transmission ratio made it possible to transform the present-day torque-velocity profile of our application to the torque-velocity profile required by the motor with the new transmission ratio. Using knowledge of the estimated transmission ratio of the BiOM, 220:1, and the transmission ratio of the new design in the peak power range, approximately 20:1, a transformation ratio of 11 was determined for estimating scaling up the torque and scaling down the velocity of the motor.

### 3.4 Motor Optimization Parameters

As labeled in Figure 3.10, there were five parameters chosen for inputs to the optimization. These are $n_l$, the number of laminations, $t_m$, the magnet thickness in the circumferential
direction, $r_{po}$, the radial extent of the rotor poles, $r_{ci}$, the inner radius of the coil, and $w_{mr}$, the width of the magnet ring in axial direction.

In the optimization, it was chosen to create a mapping between $w_p$, the pole width, and $n_l$, the number of laminations, in order to drive the motor dimensions in CAD. Thus, $w_p$ was used instead for the optimization input, though $n_l$ was the parameter being optimized. This step was also a requirement in order to utilize the standard optimization method, since integer programming does not have a wide array of packages readily available.

Figure 3.10: Single-Phase Pi-Section with Labeled Optimization Parameters. Five inputs were chosen as inputs for the optimization, $n_l$, the number of rotor pole laminations, $t_m$, the magnet thickness, $r_{po}$, the rotor pole radial extent, $r_{ci}$, the inner radius of the coil, and $w_{mr}$, the magnet ring width.
3.5 Robustness of Pi-Section Evaluations Independent of Pi-Section Cut Angle

In order to substantially decrease the simulation time, it was chosen to simulate a circumferentially symmetric model of the transverse flux motor and to take advantage of the symmetry in the simulation. See Figure 3.12 for an example of the pi-section used. This allows for a reduction in simulation time on the order of the number of pole pairs, which was in our situation 20.

To verify that the simulation results using a pi-section would be sufficiently independent of the pi-section extrusion angle, several simulations were run with various pi-section extrusion angles, and the torques and power losses were compared.

There was not a significant difference in the outputs between the different pi-section extrusion angles tested. This robustness indicated that no additional research was needed to determine the angle of extrusion used in simulation, and that a pi-section indeed would be of more accuracy than a full model for simulation of a circumferentially symmetric motor design.

3.6 Non-Generalizability from One Phase in Simulation to the Characteristics of a Three-Phase Motor

Further reduction of the motor simulation time by a factor of 3 by generalization of the results of one phase to the results of three phases was also investigated. Comparing the torque waveforms between the three phases was sufficient alone to determine that the results of one phase do not generalize trivially to the results of the two other phases. This comparison was accomplished through plotting the first waveform and copies of the second and third waveforms shifted by 120° (in the directions of their respective offsets) and inspecting the differences and percent errors from the first to the second and from the first to the third waveform (see Figure 3.11 for this analysis).

Consequently, it was determined that a three-phase pi-section should be used in all simulations. Figure 3.12 shows an example of this motor pi-section as displayed in the magnetic modeling software.
Figure 3.11: Comparison Between Torque in Three Phases. Subfigure (a) shows the torque from each of the three phases of the motor over one electrical cycle, with the second and third phases shifted to match the first phase torque. Subfigure (b) shows the difference between the shifted torque waveforms and the torque in the first phase. Subfigure (c) displays this difference as a percent error in predicting the second and third phase torque waveforms from the first phase torque. Note that the error is non-negligible.

3.7 Determination of Simulation Step Size and Mesh Element Size

Because simulation time is among the foremost costs in the design of a transverse flux motor, additional effort was spent to ensure that a fair trade-off between simulation time and accuracy was met. Though the transverse flux motor topology can be understood and analyzed in two-dimensions, accuracy in optimizing the three-dimensional structure of the motor requires three-dimensional magnetic modeling. Simulating even a pi-section of the motor requires several days, if not weeks, when too many steps are utilized and the mesh element size is too small.

Because the mesh size in the air gap relates most substantially to torque calculation, the mesh outside of the air gap was allowed to auto-generate on each step, while the mesh in the air gap has been optimized to minimize simulation time and retaining maximum accuracy. Using the assumption that simulation time is directly proportional to number of mesh elements and that the number of mesh elements is inversely proportional to mesh element size, convergence of torque values was approached beginning with a large mesh element size and decreasing along an inverse curve towards a mesh element size smaller than the air gap itself. This allowed the plotting of a Pareto front displaying the trade-off between accuracy and simulation time, and a mesh element size was chosen below which
the improvements in accuracy were insufficient to justify the added simulation time. See Figure 3.13 for these results.

Multiple step sizes with common factors were evaluated so that the steps sharing time values could be compared. It was determined that the step size has little effect on the accuracy of the torque calculation in the magnetic modeling results at each point of evaluation (see Figure 3.14). However, the importance of sufficient information about torque harmonics requires careful selection of a step size. Analyzing the torque waveform with a small step size allows for determination of the frequency content, allowing for an informed decision of the largest step size which will capture sufficient torque information to calculate average torque and torque ripple.

As a result of the step size and mesh element size optimization, the time for a single case was reduced a little over an hour, allowing the simulation and characterization of one motor configuration every four hours, or six optimization function evaluations per day.
Figure 3.13: Pareto Frontier for Various Mesh Element Sizes. Using the smallest air gap mesh size as a the truth value, torques from simulations using various air gap mesh sizes were compared, and the results are shown in Subfigure (b). Subfigure (a) shows the pareto frontier trading off error and simulation time. An airgap mesh size of 0.21 mm was chosen, seeing that the returns were diminishing quickly beyond this size.

Figure 3.14: Torque Over Time Given Fine and Rough Step Sizes and Error at Shared Steps. Using a fine step size that was a multiple of a rough step size, two simulations were compared, and the results are shown in Subfigure (a). Error as a difference is shown in Subfigure (b). Noting the size of the units in the error plot in comparison with the torque amplitude in the torque plot, it can be concluded that any differences due to step size are negligible, and that the main importance of step size is in capturing frequency information.

3.8 Pole Iron Loss Approximation

As seen in Figure 3.2, the transverse flux motor design we optimized has laminations that wrap around the radially outermost tips of the rotor poles. Magnetic flux interacts in a complex manner with these rotor pole tips as flux travels from stator north poles to stator south poles crossing a mutual air gap. Though the ideal behavior of these flux paths is up,
around, and back down the lamination, following the lamination tangentially, any normal component of the flux will cause an eddy current inside the lamination.

The magnetic modeling software we utilized contains several methods for calculating lamination iron losses via approximation methods. However, for the reasons stated above, the validity of the magnetic modeling lamination iron loss method in the rotor poles of our specific motor topology was questioned. To test the accuracy of these approximation methods, a motor design was simulated using the geometry with the approximation methods (without the laminations designed into the SolidWorks CAD model). Then, the same design was created again with the lamination sheets represented by the CAD model itself, previous to importing the model into the magnetic modeling software. The results of the design with the laminations were held as the truth values, and the results were compared. The results of the design using the approximation methods were found to be off by an order of magnitude for all of the methods tested in the software.

Having determined that the approximation methods were insufficient for characterization of the iron losses in the poles, alternative solutions were considered. Because characterization of a single motor geometry configuration using laminations enforced via CAD model was found to take on the order of weeks to finish, a simpler approximation was settled upon. The number of laminations were varied from a baseline design, and then the pole radial extent was varied from this same baseline design, in both cases using laminations enforced via CAD model in the simulations. The pole iron loss results of these simulations were compared in order to determine a percentage difference equation dependent on the pole radial extent and pole width variation from this baseline model, and this percentage difference was used to modify $k_{\text{iron}}$, the coefficient of the pole iron loss equation. See Figure 3.15 for a plot of the percentage difference relationship of various pole radial extents and pole widths. This approximation was used only for the poles and not for the secondary iron losses (those in the flux return, magnets, and flux directors), which have structures typical of traditional motor designs, and as such were simulated with confidence in the software.

3.9 Simulation of One-Half an Electrical Cycle

The determination to not use the approximation methods for rotor pole iron loss calculations allowed for one additional reduction in simulation time. Whereas the iron loss approximation methods for rotor laminations required an entire electrical cycle, all other necessary results did not. Taking advantage of motor symmetry, it appeared that the first
Figure 3.15: Pole Iron Loss Approximation as a Function of Pole Radial Extent and Pole Width. Due to the excessive length in simulation time, it was necessary to simulate over a range of pole radial extents and pole widths and create an approximation. The z-axis of this plot is percent difference from baseline, and is used for modifying the iron loss constant for the pole iron loss.

half of the electrical cycle in the motor could be used to predict the second half of the electrical cycle. In order to test the validity of this assumption, a simulation was run for a full electrical cycle at the three speeds and torques used for motor characterization. For each phase, the voltage from the second half and the voltage negated from the first half were plotted and compared. This same exercise was performed with torque and loss (noting here that this loss did not include pole iron loss, since this was computed separately), though without negating the first half results. See Figure 3.16 for these comparisons from the lowest speed and lowest torque. Due to the nature of the percent error calculation (prediction error divided by actual result), these errors were greatest for this low-speed low-torque case.

For the voltage, the error was computed as the difference between the maximum absolute values of the voltages for each phase, divided by the maximum absolute value of the
simulated second half cycle, since the maximum voltage amplitude is the result we were concerned with, as

\[ V_{\text{error}_i} = \frac{|V_{i_1}|_{\text{max}} - |V_{i_2}|_{\text{max}}}{|V_{i_2}|_{\text{max}}} \]  

(3.12)

where \( i \) is the phase number. For phases one, two, and three, the percentage errors for this metric were 0.18%, 0.18%, and 0.19%, respectively.

For the torque, the error was computed as the difference between the average torques for each phase divided by the average torque of the simulated second half cycle, since average torque is the results we desired, as

\[ \tau_{\text{error}_i} = \frac{|\tau_{avg_i_1} - \tau_{avg_i_2}|}{\tau_{avg_{i_2}}} \]  

(3.13)

For each of the respective phases, the average torque errors were 1.69%, 0.87%, and 0.16%, respectively.

Power loss error was computed similarly, but for the total loss (since the phases share flux paths and hence cannot truly be separated in their iron loss evaluation), as

\[ p_{\text{losserror}} = \frac{|p_{\text{lossavg}_1} - p_{\text{lossavg}_2}|}{p_{\text{lossavg}_2}} \]  

(3.14)

where the error was found to be 0.04%.

3.10 Calculation of Dependent Variables

For the flux return, an approximation of the packing factor was used given the lamination thickness and lamination spacing. This was a function of the thickness of the flux return, which was dependent upon the coil inner radius input.

Resistance was calculated using copper as the wire material. A constant number of turns of 21 was used, selection by experimentation to bring the initial current and voltage requirements into reasonable ranges with respect to the controller and with respect to one another.

The rotor inertia and rotor mass were calculated by equation using the dimensions and positions of each part as a function of the input vector. Conservative estimates for geometry calculations were used when the error introduced was negligible.
Figure 3.16: Second Half Simulated and Predicted Voltage, Torque, and Loss. Subfigures (a), (b), and (c) show a comparison of voltage, torque, and loss, respectively, predicted vs. simulated for the second half of an electrical cycle. This exercise proves that the first half of the electrical cycle is of sufficient information to provide all needed results for the optimization.

3.11 Choice of Boundaries

To choose the boundaries on magnet thickness and coil inner radius, the geometric parameters were varied until the CAD model was close to breaking. For both cases, this included zero volume geometry limits. In the case of magnet thickness, this also included the part interference limit, as well as a limit beyond which the pi-section would slice the part at additional points, creating an error in JMAG. After finding these limits, bounds within these limits were chosen as the constraints. For the pole radial extent, the upper limit was selected, based upon design constraints, as the radius of the motor, and the lower limit was arbitrarily selected to be far below an expected reasonable value. The magnet ring width was chosen similarly, with the upper limit being dictated by the length of the motor, and the lower limit being chosen as a value which was too small to be a design of expected usefulness.

The number of laminations selection was more involved than the other constraints. The number of laminations was ultimately chosen to be 5, 6, 7, 8, or 9. This choice was fundamentally guided by the setup of the geometry in the CAD file. Because of the pleated nature of the poles, in increasing the number of laminations (or equivalently, the pole width), there is always a point at which the inner radius (side farther from axis of motor) of the inner pleat folds in upon itself, requiring infinite curvature, or a sharp point. Not only is this not
physical, but it also breaks the inner radius arc sketch element in the CAD model.

In order to compensate for this, the inner radius of the entire pole section was set up so that it would expand outwards as the pole width expanded. This relationship was determined by experimentation, and the relationship was enforced through a sketch that “implemented” the trigonometric function using a carefully selected triangle geometry. By pushing the inner pleat radially outward when there are more laminations, the curvature of the inner radius of the inner pleat was reduced, avoiding the CAD model conflict. The number of laminations was constrained to be within the numbers of laminations which allowed all laminations to be in contact with the flux return when the coil inner radius is set to the baseline value.

Beside these geometry input constraints, the voltage waveform was also constrained to lie within the maximum possible voltage of the prosthesis controller that was recently developed in our lab for prosthesis design and control.
Chapter 4

Evaluation, Analysis and Comparisons

4.1 Optimization Results

Figures 4.1, 4.2, 4.3, and 4.4 contain the optimization progress and results. Notice that Figure 4.1 shows voltage constraint violations for 7 of the 8 first steps before the algorithm directs the inputs away from the high required voltage space. As the space is searched, the average power loss of the best design is gradually reduced.

Figure 4.2 displays the six-dimensional data on one plot, with each of the vertices representing a corresponding geometry dimension as a percent of the bounds of that geometry dimension, and the color representing the average power loss corresponding to the geometry combination of those vertices.

Figure 4.3 shows scatter plots of the power loss with each of the dimensions of the corresponding configuration. With sufficient data, one begins to see rough correlations, though this visualization tool is incomplete without the combined data of the other plots.

Finally, average power and motor mass are plotted against one another in Figure 4.4 for a further understanding of the trade-off between prosthesis mass and efficiency.

4.2 Comparison Between Motors

The Maxon EC30 Powermax brushless DC motor, as shown in Figure 4.5, is the motor currently employed in the PowerFoot BiOM. The squared torque copper loss proportionality constant of this motor is approximately 505.796 W/(N-m)^2. Assuming that the iron losses are negligible (a reasonable assumption for a brushless DC motor) and running the motor through the power calculation over one gait cycle in our selected application, we
Figure 4.1: Five Parameter Optimization Line Plots. The first plot shows the output of the cost function, which is average power loss. The next five plots show the progress of the motor configuration. Maximum voltage, also constrained is included a a subplot as well. Maximum current and efficiencies are included for reference. Orange triangles signify constraint violations.
Figure 4.2: Five Parameter Optimization Andrew’s Curves. Each curve corresponds in shape to a particular motor configuration and in color to the average power loss of that configuration over a single gait cycle.

find that the average power loss is about 27.8 W using a transmission ratio of 220:1, the configuration in which the actuator is currently assembled.

The average power output of the ankle over this gait cycle is 32.0 W, bringing the total electrical power input required to 59.8 W for the Maxon motor system. Using a LiPo battery with 6 cells of nominal voltage 3.7 V each and battery capacity 1.4 A-hrs, the energy storage of the power source totals 6 cells · 3.7 V · 1.4 A-hr · 3600 s/hr, or 111,888 Joules. This means that the total walking time on this speed with this battery and this actuator configuration is 1871 seconds, or 31 minutes, 11 seconds.

Though the walking speed of the subject from which the bionic ankle torque-velocity data was obtained was not known, noting that the gait cycle period was 1.05 seconds and extrapolating the cadence and nominal speed data from Orendurff et. al.[13] gave an estimated walking speed of 1.75 m/s. At this speed the 268 pound subject would cover a distance of 3.27 km, or 2.03 miles, on one battery charge.
Figure 4.3: Five Parameter Optimization Scatter Plots. These scatter plots give a relationship between the average power loss and each of the variables independently.
Figure 4.4: Mass Vs. Power Loss for the Transverse Flux Motor Designs. This plots shows the trade-off that inherently exists in motor design between mass and efficiency, but for the transverse flux motor.

The unoptimized transverse flux motor (of the configuration chosen for the initial geometry) has an average power loss of 11.6 W over the gait cycle. By similar calculations, the required power input with this size loss is 43.6 W, the total time is 42 minutes, 44 seconds, and the total distance is 4.48 km, or 2.78 miles, on one battery charge.

After 54 iterations of the optimization, the transverse flux motor loss decreased to 9.9 W, with a working time of 44 minutes and 33 seconds, and a total distance of 4.67 km, or 2.90 miles on a single battery charge.

For a summary of these results, see table 4.1. Note that the torque-velocity curve used here already includes the work required to account for power loss in the ball screw.
Table 4.1: Comparison of Motor Implementations in the Bionic Ankle

<table>
<thead>
<tr>
<th>Motor</th>
<th>$p_{loss}$ (W)</th>
<th>$p_{in}$ (W)</th>
<th>$t$ (min)</th>
<th>$d$ (km)</th>
<th>$m$ (kg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maxon EC-30 Powermax</td>
<td>27.838</td>
<td>59.811</td>
<td>31.180</td>
<td>3.270</td>
<td>0.271</td>
</tr>
<tr>
<td>Transverse Flux Initial Guess</td>
<td>11.663</td>
<td>43.636</td>
<td>42.740</td>
<td>4.482</td>
<td>0.569</td>
</tr>
<tr>
<td>Transverse Flux Optimized</td>
<td>9.885</td>
<td>41.857</td>
<td>44.550</td>
<td>4.672</td>
<td>0.556</td>
</tr>
</tbody>
</table>
Figure 4.5: The Maxon EC30 Powermax is the motor presently employed in the PowerFoot BiOM. *Image Credit: Maxon Motor*

### 4.3 Advantages and Hurdles in Using the Transverse Flux Motor

From Section 4.2, it is easily apparent that the transverse flux motor carries substantial efficiency advantages. From Table 4.1 it can be seen that the bionic ankle walking range on a single battery charge can be increased by 40% when the motor is replaced by a transverse flux motor.

It is noted that reluctance machines, to which class the transverse flux motor belongs, are often very noisy. However, it is believed that the motor optimization package presented in this thesis is a solution to the noise reduction from the motor. The high efficiency of the transverse flux motor while operating at low speeds and high torques can then be used to reduce the transmission ratio (as was performed in the simulations discussed here), resulting in a significant decrease in acoustic output. See Section 5.2 for more considerations surrounding cogging torque reduction.

It has become clear from research into the transverse flux motor that it is not a simple design to manufacture. New design processes must be developed before the transverse flux motor can be constructed on a miniature scale, and these processes then must be altered to use the motor at scale. However, we look forward with anticipation to the development of these new techniques, as well as to the testing of prototypes in the context of the bionic
ankle.
Chapter 5

Future Work

5.1 Construction of the Actuator

With these simulation results informing our design, we are working with Planet Rider LLC to construct the compact three-phase transverse flux motor. Once constructed, the motor will be tested and the motor will be characterized and compared with the characterization calculated in simulation. This motor prototype will be implemented in the bionic ankle and tested for efficiency and control.

5.2 Improvement of the Theory

The understanding gained from the construction and testing of the transverse flux motor will be used to further improve the models used to characterize the motor in simulation.

Additional improvements to the design can still be made in simulation. A method of correcting voltage waveforms by phase shifting of the phases mechanically appears to also be successful at reduction of cogging torque. This method would slightly offset the rotor poles in order to smooth the voltage and torque waveforms. Future work in using this optimization package would be well-invested in optimizing a combination of phase shifting between the rotor poles. Because the topology used in this thesis was a fundamental design without any of this phase shifting, the cogging torque was not considered in the optimization in order to not impose an additional restriction which was expected to be removed in future iterations of the motor. This element of the motor design has important implications in the acoustic output of the motor itself, and thus should be the next focus of attention.
It may also be considered in future implementations of the software package to use more points in the characterization in order to more confidently determine the curve fits and to consider variables in addition to the ones chosen as inputs. For instance, the motor length could be kept constant while varying a trade-off between pole axial thickness and the width of the coil.

Further, the dimensionality of pole iron loss, along with the strength of dependence of the pole iron loss on each dimension in the transverse flux motor, has not been fully explored. This may eventually be an area of investigation worth further research and characterization.

Finally, it is of important note that the design technique set forth herein is generally applicable to many types of motors. It may be of consideration to apply the same techniques set forth in this thesis to the design of actuators using multiple possible motor types.

### 5.3 The Future of Mobility and a New Class of Machines

The great importance of this work lies not in development of this one prosthesis subject to the torque-velocity profile of the gait cycle considered. Rather, these motor optimization methods lay the groundwork for optimized motors for an array of applications. Immediately relevant is the notion of tuning motor designs to individual amputees, taking into account varying weights, self-selected walking speeds, and even preferred variations in gait cycle between prosthetic users.

Also directly relevant is the extension of this technology to exoskeletons for orthotic, rehabilitative, and augmentative applications. In the next several decades, as the human population begins to increase in exoskeleton use, there are not only efficiency and power considerations, but also serious acoustic limitation requirements. To push mobility solution technologies such as prostheses and exoskeletons to the rest of earth’s population without reduction in acoustic output would be similar to having smartphones as loud as 20th century computers in every person’s pocket.

These same principles are applicable more generally to a new emerging class of machines, ones which operate quietly and efficiently. One can imagine with some effort quiet garbage disposals, refrigerators, microwaves, and vacuums, as well as robotic servants working quietly through the night to tidy the house, making only just enough noise in the morning to wake you to the sound of breakfast being prepared. Further, as urbanization increases, demand for quiet lift and air transportation solutions, as well as noise reduction in construction methods, will increase significantly.
Further, many devices are emerging with cyclic variable torque-speed profiles and high efficiency requirements. The recent development of the MIT Cheetah is one example application. Additional such applications include development of artificial hearts, robotics which swim quietly using a tail, climbing robots, and robots with actuating wings. More personal applications are emerging, with the development of neural control suggesting the future need for specialized prosthetic arms and hands which are tuned for specific tasks, such as tennis, typing, or car repair.

It is the author’s hope that this software package will empower designers in the development of these technologies, as well as technologies beyond those which have yet been conceived, in making our world more accessible, more enjoyable, and more rewarding for everyone living in it now and in the future.
Appendix A

Calculation of Dependent Variables

This section outlines the dimensions that were used for the transverse flux motor optimization. The motor dimension variables were dimensions that were be varied. Dependent variables were the variables which relied on the variable dimensions. Independent motor dimension constants were be altered, so dependent motor dimension constants also remained fixed. Motor materials were not varied. The constants in the optimization were held constant for simplicity in the optimization, since an increase in the number of input dimensions increases time needed for convergence of the optimization.

Since SolidWorks was used and is case-insensitive, great care was taken to choose variable names which were case-insensitive. For this reason, radii were subscripted “o” for outer and “i” for inner. Note that for simplicity no underscores were used in the code.

Variables used to dynamically modify the SolidWorks geometry are colored red. Variables used to dynamically modify JMAG parameters are colored blue. The rotor inertia is colored green, and the motor mass is colored magenta.

A.1 Motor Dimension Variables

The variables mentioned above have the following baseline values:

Number of Laminations: $n_l = 7$

Magnet Thickness (Circumferential/Magnetization Direction): $t_m = 1.85 \text{ mm}$

Pole Radial Extent: $r_{po} = 30.125 \text{ mm}$

Coil Inner Radius: $r_{ci} = 14.725 \text{ mm}$
Magnet Ring Width: \( w_{mr} = 6 \text{ mm} \)

**A.2 Independent Motor Dimension Constants**

The following values will be held constant during the optimization:

- Motor Outer Radius (Radial Distance to Magnet Top Corner): \( r_o = 31 \text{ mm} \)
- Motor Inner Radius: \( r_i = 10 \text{ mm} \)
- Number of Pole Pairs: \( p = 20 \)
- Magnet Height (Radial Direction): \( h_m = 11.125 \text{ mm} \)
- Flux Concentrator Inner Radius: \( r_{fi} = 20.5 \text{ mm} \)
- Flux Concentrator Outer Radius: \( r_{fo} = 30.375 \text{ mm} \)
- Coil Outer Radius: \( r_{co} = 19.825 \text{ mm} \)
- Flux Concentrator to Pole Air Gap: \( g_{fp} = 0.125 \text{ mm} \)
- Return to Coil Air Gap: \( g_{rc} = 0.350 \text{ mm} \)
- Pole Thickness (Axial Direction): \( t_p = 5.5 \text{ mm} \)
- Lamination Thickness: \( t_l = 0.2 \text{ mm} \)
- Lamination Spacing: \( t_s = 0.05 \text{ mm} \)
- Number of Coil Turns: \( n_c = 21 \)
- Leakage Inductance: 0 H
- Maxon Rotor Inertia (for replacement): \( 3.33 \cdot 10^{-6} \text{ kg}\cdot\text{m}^2 \)

**A.3 Motor Material Constants**

**Armature**

Material: Soft Magnetic Material > Steel Sheet > JFE Steel > 20JNEH1200
Density: $\rho_p = 7650 \text{ kg/m}^3$
Resistivity: $\rho_{jn} = 550 \cdot 10^{-9} \text{ } \Omega\text{m}$

**Magnets**

Material: Permanent Magnet > Sintered NdFeB >
Hitachi Metals(Formerly SSMC) > Irreversible > NEOMAX-39SH
Density: $\rho_m = 7270 \text{ kg/m}^3$
Resistivity: $\rho_{nd} = 1.6 \cdot 10^{-6} \text{ } \Omega\text{m}$
Remanence: $B_r = 1.329 \text{ T}$

**Flux Concentrators**

Material: Soft Magnetic Material > Soft Magnetic Composite >
HoganasAB > Somaloy_700_3P
Density: $\rho_f = 7450 \text{ kg/m}^3$
Resistivity: $\rho_{so} = 280 \cdot 10^{-6} \text{ } \Omega\text{m}$

**Coil**

Material: Conductor > JSOL > Copper
Density: $\rho_c = 8960 \text{ kg/m}^3$
Resistivity: $\rho_{cu} = 1.7 \cdot 10^{-8} \text{ } \Omega\text{m}$

### A.4 Dependent Motor Dimension Constants

Return Inner Radius: $r_{ri} = r_i = 10 \text{ mm}$

Half Pole-Pitch: $\theta_{hp} = \frac{2\pi}{2p} = 9^\circ$

Radial Distance to Magnet Top Center: $r_{mo} = \sqrt{r_o^2 - \left(\frac{t_m}{2}\right)^2} \approx 30.986 \text{ mm}$

Radial Distance to Magnet Bottom Center: $r_{mi} = r_{mo} - h_m \approx 19.861 \text{ mm}$

Coil to Magnet Minimum Gap: $g_{cm} = r_{mi} - r_{co} \approx 0.036 \text{ mm}$

Pole Inside Arc Arc Angle: $\theta_a = \frac{\pi}{2} - \theta_{hp} = \frac{9\pi}{20} \text{ radians}$

Pole Inside Arc Corresponding Isosceles Angles: $\theta_b = \frac{\pi - \theta_a}{2} = \frac{11\pi}{40} \text{ radians}$
A.5 Dependent Motor Dimension Variables

Coil Inner Diameter: \( d_{ci} = 2r_{ci} \)

Flux Concentrator Width (Axial Direction): \( w_f = w_{mr} \)

Magnet Width (Axial Direction): \( w_m = w_{mr} \)

Coil Width (Axial Direction): \( w_c = w_{mr} \)

Return Width (Axial Direction): \( w_r = w_{mr} + 2g_f \)

Linear Pattern Distance: \( d = t_p + w_r \)

Double Linear Pattern Distance: \( dd = 2d \)

Return Outer Radius: \( r_{ro} = r_{ci} - g_{rc} \)

Return Outer Diameter: \( d_{ro} = 2r_{ro} \)

Coil Packing Factor: \( \lambda_c = 0.9 \)

Coil Resistance (inputs in meters): \( r_1 = \frac{\rho_{cu} L_c}{\lambda_c A_c} = \frac{\rho_{cu} 2\pi r_{co} + r_{ci} \pi}{\lambda_c (r_{co} - r_{ci}) w_{mr}} \)

Pole Width (Circumferential Direction): \( w_p = 2n_t t_t + (2n_t - 1) t_s \)

Half Pole Width: \( w_{hp} = \frac{w_p}{2} \)

Pole Inside Arc Inner Radius: \( r_{ai} = 1.3455 \text{ mm} - w_{hp} \cos 54.5495^\circ \)

Pole Inside Arc Outer Radius: \( r_{ao} = r_{ai} + w_{hp} \)

Mean Pole Inside Arc Length: \( l_a = \frac{r_{ao} + r_{ai}}{2} \)

Radial Distance to Pole Inside Arc Center: \( r_{ac} = \frac{w_{hp} + r_{ai}}{\sin \theta_{hp}} \)

Pole Pleat Inner Radius: \( r_{pi} = r_{ac} - (w_{hp} + r_{ai}) \)

Radial Distance to Pole Rectangular Section Bottom Center: \( r_{pb} = r_{ac} \cos \theta_{hp} \)

Radial Distance to Pole Rectangular Section Top Center: \( r_{pt} = r_{po} - w_{hp} \)

Pole Rectangular Section Height: \( h_p = r_{pt} - r_{pb} \)

Pole Top Center of Mass: \( c_1 = (0, r_{po} - \frac{w_{hp}}{2}) \)
Pole Rectangular Section Center of Mass: \( c_2 = (0, \ r_{pt} - \frac{b_p}{2}) \)

Pole Half Inside Arc Center of Mass:
\[
c_3 = \left( \frac{la}{2} \cos \theta_b + \frac{w_h}{2} \sin \theta_b, \ r_{pb} - \frac{la}{2} \sin \theta_b + \frac{w_h}{2} \cos \theta_b \right)
\]

Single Pole Top Mass: \( m_{p1} = \rho_{p1} \pi w_{hp}^2 \)

Single Pole Rectangular Section Mass: \( m_{p2} = \rho_{p2} w_{ph} \)

Half Pole Inside Arc Mass: \( m_{p3} = \rho_{p3} \frac{th}{2} (r_{ao}^2 - r_{ai}^2) \)

Return Mass: \( m_r = \rho_{pr} w_{r} \pi (r_{ro}^2 - r_{ri}^2) \)

Single Pole Top Inertia (High Approx.): \( I_{z_{p1}} = \rho_{p1} \frac{w_{hp}^2 + \frac{w_h^2}{3}}{12} + m_{p1} c_{1y}^2 \)

Single Pole Rectangular Section Inertia: \( I_{z_{p2}} = \rho_{p2} \frac{w_{ph}^2 + \frac{w^2}{3}}{12} + m_{p2} c_{2y}^2 \)

Half Pole Inside Arc Inertia (High Approx.): \( I_{z_{p3}} = \rho_{p3} \frac{w_{hp}^2 + \frac{w_h^2}{3}}{12} + m_{p3} (c_{3x}^2 + c_{3y}^2) \)

Return Inertia: \( I_{zr} = \rho_{pr} w_{r} \pi (r_{ro}^4 - r_{ri}^4) \)

Three-Phase Rotor Inertia (High Approx.): \( I_z = 4\rho (I_{zp1} + I_{zp2} + 2I_{zp3}) + 3I_{zr} \)

Magnet Mass: \( m_m = \rho_m w_m h m \)

Flux Concentrator Mass (High Approx.): \( m_f = \rho_f w_f \frac{1}{2} (\frac{th}{r_{fo}} - \frac{t_m}{r_{fo}}) (r_{fo}^2 - r_{fi}^2) \)

Coil Mass: \( m_c = \rho_c w_c \pi (r_{co}^2 - r_{ci}^2) \)

Three-Phase Motor Mass (High Approx., Electromagnetics Only):
\[
m = 4\rho (m_{p1} + m_{p2} + 2m_{p3}) + 3(m_r + m_c + 2p(m_m + m_f))
\]

Pole Packing Factor: \( \lambda_p = \frac{2n_{l1}}{w_p} \)

Thickness of Return: \( t_r = r_{ro} - r_{ri} \)

Number of Return Laminations: \( n_{lr} \approx \frac{t_r - \frac{1}{2}t_{li}}{t_{li} + t_r} + 1 \)

Return Packing Factor: \( \lambda_r \approx \frac{n_{lr} t_{li}}{n_{lr} t_{li} + (n_{lr} - 1)t_s} \) for \( n_{lr} \in \mathbb{R} \mid n_{lr} \geq 2 \)
Appendix B

Motor Optimization Code

```
############ motorOptimizationPackage.py ############
# PACKAGES
import numpy as np
import pandas as pd
import scipy.optimize
import os, sys
import calculateDependentVariables as CDV
import setupAndRunSimulation as SARS
import processCsvSimulationResults as PCSR
import calculatePoleIroLossConstants as CPLC
import calculateAveragePowerLoss as CAPL
#
# CHECK COMPLETED EVALUATIONS
#
def checkCompletedEvaluations(motorParams, optimizationLogFilepath):
    # Check that the file exists and get dataframe
    if not os.path.isfile(optimizationLogFilepath):
        colNames = ['motorModel', 'returnedPLossAvg']
        dimCol = ['dim{0}'.format(d+1) for d in range(len(motorParams))]
        colNames.extend(dimCol)
        logDF = pd.DataFrame(columns=colNames)
        logDF.to_csv(optimizationLogFilepath)
        # Haven't run anything yet, so return None
        return None
    # Create a difference threshold for comparison of input with
    # completed function evaluations.
    eps = 1e-12
    # Read in Log file
    logDF = pd.DataFrame.from_csv(optimizationLogFilepath)
```
dimCol = [c for c in logDF.columns.values if "dim" in c]
for i in logDF.index:
    prevMotorParams = np.array(logDF.ix[i][dimCol].tolist())
    prevPLossAvg = logDF.ix[i]['returnedPLossAvg']
    if a match is found,
        if (abs(motorParams - prevMotorParams) < eps).all():
            return prevPLossAvg
    # If nothing was found, return None.
    return None

# ADD NEW ROW TO LOG
def addNewRowToLog(optimizationLogFilepath, motorModelString, returnedPLossAvg, motorParams):
    # Load current optimization log
    logDF = pd.DataFrame.from_csv(optimizationLogFilepath)
    colNames = logDF.columns.values
    # Create DataFrame with new row
    row = [motorModelString, returnedPLossAvg]
    row.extend(motorParams)
    newDF = pd.DataFrame([row], columns=colNames)
    # Append the new row to the log
    logDF = logDF.append(newDF, ignore_index=True)
    # Write log back to csv
    logDF.to_csv(optimizationLogFilepath)
    return

# GET AVERAGE POWER LOSS
def convertParam(optimizationValue, motorBounds, optimizationBounds):
    minMotor = min(motorBounds)
    maxMotor = max(motorBounds)
    minOptimization = min(optimizationBounds):
    optimizationBounds = [1.0, 11.0/9.0]
    bounds = {"wp": [2.45, 4.45],
              "tm": [1.5, 2],
              "rpo": [28.0, 31.0],
              "rci": [13.5, 17],
              "wmr": [5, 6.4]}
    wp = convertParam(x[0], bounds["wp"], optimizationBounds)
    tm = convertParam(x[1], bounds["tm"], optimizationBounds)
    rpo = convertParam(x[2], bounds["rpo"], optimizationBounds)
    rci = convertParam(x[3], bounds["rci"], optimizationBounds)
    wmr = convertParam(x[4], bounds["wmr"], optimizationBounds)
    return [wp, tm, rpo, rci, wmr]
```python
def getAveragePowerLoss(x):
    """Get the average power loss over a single gait cycle for a given geometry."""
    motorParams = convertOptimzationParametersToMotorParameters(x)
    print "Motor-Parameters",motorParams
    # Check to see if the requested loss for the given vector was computed previously.
    p_loss_avg = checkCompletedEvaluations(motorParams, OPTIMIZATION_LOG_FILEPATH)
    # If these inputs were already run, return the appropriate p_loss_avg
    if p_loss_avg != None:
        return p_loss_avg
    # Calculate the physical motor features given the requested input geometry vector.
    features = CDV.calculateDependentVariables(wp=motorParams[0], tm=
                                            motorParams[1], rpo=motorParams[2], rci =motorParams[3], wmr=
                                            motorParams[4])
    # If the input vector is out of bounds, return a large value for the objective function.
    if features['inDesignSpace']!=True:
        addNewRowToLog(OPTIMIZATION_LOG_FILEPATH, 'outOfBounds',
                        OUT_OF_BOUNDS_PLOSS_AVG, motorParams)
        return OUT_OF_BOUNDS_PLOSS_AVG
    # If the input vector is in bounds and hasn’t been run previously,
    # Setup and run the simulation
    filepathList = SARS.setupAndRunSimulation(features, JPROJ_DIRECTORY,
                                              OUTPUT_DIRECTORY)
    # When the simulation is done, process the csv simulation results to get the loss constants.
    processedResults = PCSR.processCsvSimulationResults(filepathList)
    # Check if voltage is within design constraints
    if processedResults['maxVoltageAmplitude'] > MAX_VOLTAGE_CONSTRAINT:
        motorModelString = os.path.basename(filepathList[0])[: -9]
        addNewRowToLog(OPTIMIZATION_LOG_FILEPATH, motorModelString,
                        OUT_OF_BOUNDS_PLOSS_AVG, motorParams)
        return OUT_OF_BOUNDS_PLOSS_AVG
    # Use the geometry to calculate the pole iron loss constants.
    poleIronLossResults = CPLIC.calculatePoleIronLossConstants(features)
    # Calculate the average power loss using the loss constants and the transformed torque-speed curve.
    p_loss_avg = CAPL.calculateAveragePowerLoss(features,
```

processedResults, poleIronLossResults)
# and to the log file of completed function evaluations
motorModelString = os.path.basename(filepathList[0])[:-9]
addNewRowToLog(OPTIMIZATION_LOG_FILEPATH, motorModelString, p_loss_avg
, motorParams)
# Return the average power loss corresponding to the input vector
# geometry.
return p_loss_avg

# ———— OPTIMIZATION ————

# Static Variables
OUT_OF_BOUNDS_P_LOSS_AVG = 1e12
MAX_VOLTAGE_CONSTRAINT = 32.0
OPTIMIZATION_NAME = "motorSimOptimization4"
PATH_TO_DROBOX = "C:\\Users\\crtaylor\\Dropbox(MIT)\\motorCalculations\\motorOptimizationCode\\results"
JPROJ_DIRECTORY = os.path.join("C:\", os.sep, OPTIMIZATION_NAME)
OUTPUT_DIRECTORY = os.path.join(PATH_TO_DROBOX, OPTIMIZATION_NAME)
if not os.path.isdir(JPROJ_DIRECTORY):
    print "JMAG._Directory .. doesn’t exist .. Check .. path"
    print JPROJ_DIRECTORY
    sys.exit()
if os.path.isdir(OUTPUT_DIRECTORY):
    continue_output = raw_input("Output directory already exists .. okay to continue? (Yes/n)"
if continue_output != "Yes":
    print "Not okay to continue .. exiting"
    sys.exit()
else:
    print "Making output directory"
    os.mkdir(OUTPUT_DIRECTORY)

OPTIMIZATION_LOG_FILEPATH = os.path.join(OUTPUT_DIRECTORY, "completedEvaluations.csv")
midPoint = 10.0/9.0
x0 = np.array([midPoint, midPoint, midPoint, midPoint, midPoint])
# Optimization
xopt, fopt, iter, funcalls, warnflag, allvecs = scipy.optimize.fmin(
    getAveragePowerLoss, x0, full_output=True, retall=True)

############ calculateDependentVariables.py ############

# PACKAGES
import numpy as np
# DIMENSIONS

def calculateDependentVariables(wp=3.45, tm=1.85, rpo=30.125, rci=14.725,)
wmr=6.0):

'\'\'Given the input vector of motor dimension variables, calculate all dependent variables.'\'\'
print "Calculating Dependent Variables"
features = {} # See the resistance for support.
# ===== Motor Dimension Variables =====
if wp < 2.45 or wp > 4.45 or tm < 0.5 or tm > 2 or rpo < 28 or rpo > 31 or rci < 13.5 or rci > 17 or wmr < 5 or wmr > 6.4:
features["inDesignSpace"] = False
else:
features["inDesignSpace"] = True
features["wp"] = wp
features["tm"] = tm
features["rpo"] = rpo
# ===== Independent Motor Dimension Constants =====
ro = 31.0
ri = 10.0
p = 20.0
hm = 11.125
rfi = 20.5
rfo = 30.375
rco = 19.825
gfp = 0.125
grc = 0.350
tp = 5.5
tl = 0.2
ts = 0.05
nc = 23.0
# ===== Dependent Motor Dimension Constants =====
ri = ri
thetahp = (2.0*np.pi)/(2.0*p)
rmo = np.sqrt(ro**2.0-(tm/2.0)**2.0)
rm = rmo - hm
gcm = rmi - rco
thetaa = np.pi/2 - thetahp
thetab = (np.pi-thetaa)/2
# ===== Dependent Motor Dimension Variables =====
features["dci"] = 2.0*rci
wf = wmr
features["wf"] = wf
wm = wmr
features["wm"] = wm
wc = wmr

features["wc"] = wc

wr = wmr + 2.0*gfp

features["wr"] = wr

d = tp+wr

features["d"] = d

features["dd"] = 2.0*d

rro = rci - grc

features["dro"] = 2.0+rro

lambdac = 0.9

features["rl"] = (rhocu*2.0*np.pi*((rco*le-3+rci*le-3)/2.0)*nc)/((
lambdac*(rco*le-3-rci*le-3)*wmr*le-3)/nc)

nl = (wp+ts)/(2*(tl+ts))

features["nl"] = nl

whp = wp/2.0

rai = 1.3455 - whp*np.cos(54.5495*np.pi/180)

rao = rai + whp

la = thetaa*(rao+rai)/2.0

rac = (whp+rai)/np.sin(thetahp)

rpi = rac - (whp+rai)

rpb = rac*np.cos(thetahp)

rpt = rpo - whp

hp = rpt - rpb

c1 = (0.0, rpo - whp/2.0)

c2 = (0.0, rpt - hp/2.0)

features["cl"] = (0.0, rpo - whp/2.0)

features["c2"] = (0.0, rpt - hp/2.0)

features["c3"] = (la/2.0*np.cos(thetab)+whp/2.0*np.sin(thetab), rpb-la/2.0*np.

sin(thetab)+whp/2.0*np.cos(thetab))

mp1 = rhop*(tp*le-3)*np.pi*(whp*le-3)**2.0

mp2 = rhop*(tp*le-3)*np.pi*(hp*le-3)**2.0

mp3 = rhop*(tp*le-3)*thetaa/2.0*(((rao*le-3)**2.0-(rai*le-3)**2.0)

mr = rhop*(wr*le-3)*np.pi*((rro*le-3)**2.0-(rri*le-3)**2.0)

lzp1 = rhop*(tp*le-3)*((wp*le-3)**3.0*(whp*le-3)-(whp*le-3)**3.0*(wp

*le-3))/12.0+mp1*(c1[1]*le-3)**2.0

lzp2 = rhop*(tp*le-3)*((wp*le-3)**3.0*(hp*le-3)-(hp*le-3)**3.0*(wp

*le-3))/12.0+mp2*(c2[1]*le-3)**2.0

lzp3 = rhop*(tp*le-3)*((whp*le-3)**3.0*(la*le-3)-(la*le-3)**3.0*(whp

*le-3))/12.0+mp3*((c3[0]*le-3)**2.0+(c3[1]*le-3)**2.0)

lzr = rhop*(wr*le-3)*np.pi/2*(((rro*le-3)**4.0-(rri*le-3)**4.0)

features["lz"] = 4.0*p*(lzp1+lzp2+2.0*lzp3)+3.0+lzr

nm = rhom+(wm*le-3)+(hm*le-3)*tm*le-3

mf = rhof*(wf*le-3)**1/2.0*(thetahp-tm/rfo)*((rfo*le-3)**2.0-(rri*le

*le-3)**2.0)
mc = \( \rho_c \cdot (\omega_c \cdot l - 3) \cdot \pi \cdot (r_{co} \cdot l - 3) \cdot \pi \cdot \pi \cdot (r_{ci} \cdot l - 3) \) 

features["m"] = 4.0 * p * (mpl + mp2 + 2.0 * mp3) + 3.0 * (mr + mc + 2.0 * p * (mf + mf))

lambda_p = 2.0 * nl * tl / wp 

tr = rro - rri 

nlr = (tr - tl / 2.0) / (tl + ts) + 1.0 

features["lambdar"] = (nlr * tl) / (nlr * tl + (nlr - 1.0) * ts) 

return features

# setupAndRunSimulation.py

import win32com
import win32com.client.dynamic
import monitorSimulationProgress as MSP
import time
import sys
import exportSimulationResultsToCsv as ESRTC

def setupAndRunSimulation(features, jprojDirectory, outputDirectory):
    print "Establishing Connection"

    sys.stdout.flush()
    # Establish the connection to J MAG.
    app = win32com.client.dynamic.Dispatch("designer.Application")
    app.Show()
    # Get the model index for the model to be created.
    modelIndex = app.NumModels()
    # Use the model index to create a unique name for the motor model.
    modelName = u"motorModel%03d" % modelIndex
    print "Duplicating motorModel000..to..Create %s" % modelName

    sys.stdout.flush()
    # Duplicate the study
    app.GetModel(0).UpdateCadModel(True)
    # Name the new model with the unique motor model name.
    app.GetModel(modelIndex).SetName(modelName)
    print "Updating Geometry..Parameters..in..J MAG"

    sys.stdout.flush()
    cadParameters = [
        {"jmagVar":u"d","varRef":u"d@coilPattern@motorGeom3.Assembly","refKey":"d"},
        {"jmagVar":u"d2","varRef":u"d@FluxConcentratorAxialOffset1@FluxConcentrator\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d3","varRef":u"d@magnetAxialOffset1@Magnet\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d4","varRef":u"d@returnPattern@motorGeom3.Assembly","refKey":"
    
import win32com
import win32com.client.dynamic
import monitorSimulationProgress as MSP
import time
import sys
import exportSimulationResultsToCsv as ESRTC

def setupAndRunSimulation(features, jprojDirectory, outputDirectory):
    print "Establishing Connection"

    sys.stdout.flush()
    # Establish the connection to J MAG.
    app = win32com.client.dynamic.Dispatch("designer.Application")
    app.Show()
    # Get the model index for the model to be created.
    modelIndex = app.NumModels()
    # Use the model index to create a unique name for the motor model.
    modelName = u"motorModel%03d" % modelIndex
    print "Duplicating motorModel000..to..Create %s" % modelName

    sys.stdout.flush()
    # Duplicate the study
    app.GetModel(0).UpdateCadModel(True)
    # Name the new model with the unique motor model name.
    app.GetModel(modelIndex).SetName(modelName)
    print "Updating Geometry..Parameters..in..J MAG"

    sys.stdout.flush()
    cadParameters = [
        {"jmagVar":u"d","varRef":u"d@coilPattern@motorGeom3.Assembly","refKey":"d"},
        {"jmagVar":u"d2","varRef":u"d@FluxConcentratorAxialOffset1@FluxConcentrator\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d3","varRef":u"d@magnetAxialOffset1@Magnet\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d4","varRef":u"d@returnPattern@motorGeom3.Assembly","refKey":"
    
import win32com
import win32com.client.dynamic
import monitorSimulationProgress as MSP
import time
import sys
import exportSimulationResultsToCsv as ESRTC

def setupAndRunSimulation(features, jprojDirectory, outputDirectory):
    print "Establishing Connection"

    sys.stdout.flush()
    # Establish the connection to J MAG.
    app = win32com.client.dynamic.Dispatch("designer.Application")
    app.Show()
    # Get the model index for the model to be created.
    modelIndex = app.NumModels()
    # Use the model index to create a unique name for the motor model.
    modelName = u"motorModel%03d" % modelIndex
    print "Duplicating motorModel000..to..Create %s" % modelName

    sys.stdout.flush()
    # Duplicate the study
    app.GetModel(0).UpdateCadModel(True)
    # Name the new model with the unique motor model name.
    app.GetModel(modelIndex).SetName(modelName)
    print "Updating Geometry..Parameters..in..J MAG"

    sys.stdout.flush()
    cadParameters = [
        {"jmagVar":u"d","varRef":u"
    
import win32com
import win32com.client.dynamic
import monitorSimulationProgress as MSP
import time
import sys
import exportSimulationResultsToCsv as ESRTC

def setupAndRunSimulation(features, jprojDirectory, outputDirectory):
    print "Establishing Connection"

    sys.stdout.flush()
    # Establish the connection to J MAG.
    app = win32com.client.dynamic.Dispatch("designer.Application")
    app.Show()
    # Get the model index for the model to be created.
    modelIndex = app.NumModels()
    # Use the model index to create a unique name for the motor model.
    modelName = u"motorModel%03d" % modelIndex
    print "Duplicating motorModel000..to..Create %s" % modelName

    sys.stdout.flush()
    # Duplicate the study
    app.GetModel(0).UpdateCadModel(True)
    # Name the new model with the unique motor model name.
    app.GetModel(modelIndex).SetName(modelName)
    print "Updating Geometry..Parameters..in..J MAG"

    sys.stdout.flush()
    cadParameters = [
        {"jmagVar":u"d","varRef":u"d@coilPattern@motorGeom3.Assembly","refKey":"d"},
        {"jmagVar":u"d2","varRef":u"d@FluxConcentratorAxialOffset1@FluxConcentrator\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d3","varRef":u"d@magnetAxialOffset1@Magnet\"motorGeom3.Part","refKey":"d"},
        {"jmagVar":u"d4","varRef":u"d@returnPattern@motorGeom3.Assembly","refKey":"d"}
    ]
for cadParam in cadParameters:
    app.GetModel(modelIndex).GetStudy(0).AddCadParameter(cadParam["varRef"])  
    app.GetModel(modelIndex).GetStudy(0).GetDesignTable().AddCadParameterVariableName(cadParam["varRef"], cadParam["jmagVar"])
for caseIndex in [1, 2, 3]:
    app.View().SetCurrentCase(caseIndex)
for cadParam in cadParameters:
    app.GetModel(modelIndex).GetStudy(0).SetCadParameterValue(cadParam["varRef"], features[cadParam["refKey"]])
print "Updating ..SolidWorks ..Parameters"
sys.stdout.flush()
# Apply the updated geometry parameters in SolidWorks.
app.GetModel(modelIndex).GetStudy(0).ApplyAllCasesCadParameters()
print "Updating ..J MAG ..Parameters"
sys.stdout.flush()
# Update all of the J MAG condition parameters to agree with the geometry.
app.GetModel(modelIndex).GetStudy(0).GetCircuit().GetComponent(u"Coil1").SetValue(u"Resistance", features[rl])
app.GetModel(modelIndex).GetStudy(0).GetCircuit().GetComponent(u"Coil2").SetValue(u"Resistance", features[rl])
app.GetModel(modelIndex).GetStudy(0).GetCircuit().GetComponent(u"Coil3").SetValue(u"Resistance", features[rl])
app.GetModel(modelIndex).GetStudy(0).GetMaterial(4).SetValue(u"LaminationFactor", 100.0*features["lambdar"])
print "Saving ..j proj ..File"
sys.stdout.flush()
# Save the .j proj file.
app.Save()
print "Submitting ..All ..Cases ..to ..the ..Queue"
sys.stdout.flush()
# Submit all cases for the current model to the queue.
job = app.GetModel(modelIndex).GetStudy(0).CreateJob()
job.SetValue(u"Title", modelName)
job.SetValue(u"Queued", True)
job.Submit(1)
print "Monitoring Simulation Status"
sys.stdout.flush()
# Monitor the status of the simulation (and output to log file).
# For each case,
for caseIndex in [1,2,3]:
    # wait for a minute for the first case to run,
time.sleep(60)
    # assume the case has not finished,
    progress = -1
    # and while the case hasn't finished
    while (progress!=101):
        # check the progress of the simulation for that case
        progress = MSP.monitorSimulationProgress(modelName, caseIndex,
                                                   jprojDirectory)
        # and, if the simulation is in progress,
        if progress >=0:
            # wait between progress probes.
            time.sleep(10)
    # Once a case simulation is completed,
    print "%Case %d Completed" % caseIndex
    sys.stdout.flush()
print "Exporting Results to CSV"
sys.stdout.flush()
# When all three cases are completed, export the simulation results
to a csv file.
filepathList = ESRTC.exportSimulationResultsToCsv(app, modelIndex,
                                                   modelName, outputDirectory)
print "Success! The results are now in CSV files! Hooray!"
sys.stdout.flush()
return filepathList

############# monitorSimulationProgress.py #############
import os, os.path, time, sys
def s2hms(s):
    """Convert seconds to hours, minutes, and seconds"""
    # Separate seconds into minutes and seconds.
    m,s = divmod(s,60)
    # Separate minutes into hours and minutes.
    h,m = divmod(m,60)
    # Return hours, minutes, and seconds.
    return (h,m,s)
```python
def loadingBar(progress, caseIndex):
    '''Display a loading bar for the simulation progress.'''
    if progress == 101:
        progress = 100  # Just a reassignment within scope for clean display.
    # Delete the current line and replace it with the refreshed loading bar.
    sys.stdout.write('Case._.0}Simulation Progress:{1}{2}..{3}%'.
                    format(caseIndex, '#' * progress, '-' * (100 - progress), progress))
    # Display the loading bar immediately.
    sys.stdout.flush()
    return

def monitorSimulationProgress(modelName, caseIndex, jprojFolder, jprojName='motor',
                                studyName='motorStudy'):
    '''Monitor the status of a case simulation and return -1 if queued,
    percent progress if running, or 101 if completed.'''
    # Get the high-level path.
    jfilesPath = os.path.join('C:', os.sep, jprojFolder)
    # Determine the jfiles folder name from the jproj name.
    jfilesFolder = jprojName + '.jfiles'
    # Determine the model results path, where the model folder will be found.
    modelPath = os.path.join(jfilesPath, jfilesFolder)
    # and make a list of all of the folders and files sharing this path.
    modelList = os.listdir(modelPath)
    # Then get the model folder by selecting the last folder with a
    # matching model name (only works for unique model names).
    modelFolder = [i for i in modelList if modelName in i][-1]
    # Determine the study results path, where the study folder will be found.
    studyPath = os.path.join(modelPath, modelFolder)
    # and make a list of all of the folders and files sharing this path.
    studyList = os.listdir(studyPath)
    # Then get the study folder by selecting the last folder with a
    # matching study name (only works for unique study names).
    studyFolder = [i for i in studyList if studyName in i][-1]
    # Determine the case results path, where the results are contained.
    casePath = os.path.join(studyPath, studyFolder)
    # Store the case folder name,
    caseFolder = 'Case' + str(caseIndex)
    # then determine the path to this folder.
    jstage2Path = os.path.join(casePath, caseFolder)
```

# Save the paths to the two status files, jstage2 and jstage3.
jstage2 = os.path.join(jstage2Path, 'jstage2')
jstage3 = os.path.join(jstage2Path, 'jstage3')

# Initialize progress (percent complete).
progress = -2

# Check the folder to see if the status files exist.
jstage2Exists = os.path.isfile(jstage2)
jstage3Exists = os.path.isfile(jstage3)

# If jstage2 exists
if jstage2Exists:
    try:
        # from the file
        with open(jstage2, 'r') as f:
            # read the data.
x = f.readlines()

        # If the jstage2 is empty.
        if len(x)<8:
            # overwrite the data with 1's.
x = ['1', '1', '1', '1', '1', '1', '1', '1']
        except IOError as e:
            print "Attempted to open jstage2... File non-existent."
x = ['1', '1', '1', '1', '1', '1', '1', '1']

# If jstage3 also exists.
if jstage3Exists:
    try:
        # from the file
        with open(jstage3, 'r') as f:
            # read the data.
y = f.readlines()

        # If jstage3 is empty.
        if len(y)<8:
            # overwrite the data with 0's.
y = ['0', '0', '0', '0', '0', '0', '0', '0']
        except IOError as e:
            print "Attempted to open jstage3... File non-existent."
y = ['0', '0', '0', '0', '0', '0', '0', '0']

    # If jstage3 did not exist.
else:
    # write 0's to the data.
y = ['0', '0', '0', '0', '0', '0', '0', '0']

# Calculate the seconds since the simulation started.
secondsPassed = float(x[3])+float(y[3]) # Note that y[3] causes
a jump ahead in assumed progress on the first step of the simulation.

# Calculate the seconds until the simulation will be completed.
secondsRemaining = float(x[5]) - float(y[3])

# Calculate the expected time for the simulation to the completed.
secondsExpected = float(x[3]) + float(x[5])

# Get the percent done.
percentDone = float(x[7])

# Calculate the percent done using the seconds passed and total expected.
precisePercentDone = secondsPassed / secondsExpected * 100

# Convert the percent done to an integer status (JMAG rounds to the nearest integer for status).
status = int(round(precisePercentDone))

# Convert the percent done to an integer progress (JMAG truncates to an integer for progress).
progress = int(precisePercentDone)

# Otherwise, if jstage2 does not exist yet and there are less than 15 files and folders in the path,
elif len(os.listdir(jstage2Path)) < 15:
    # assign -1 to progress to indicate that the simulation has not started.
    progress = -1

# Otherwise, if jstage2 doesn't exist (and there at least 15 files and folders in the path),
else:
    # assign 101 to progress to indicate that the simulation is finished.
    progress = 101

# Display a loading bar showing the progress for the current case.
loadingBar(progress, caseIndex)

return progress

# This part only runs when this python script is run directly.
if __name__ == '__main__':
    # Store the model name to request status updates for.
    modelName = 'motorModel001'
    # Begin by monitoring case 1.
    caseIndex = 1
    # While all cases have not finished,
    while (caseIndex < 4):
        # assume the case has not finished,
progress = -1
# and while the case hasn't finished
while (progress!=101):
    # check the progress of the simulation for that case
    progress = monitorSimulationProgress(modelName, caseIndex)
sys.stdout.flush()
    # and, if the simulation is in progress,
    if progress >=0:
        # wait between progress probes.
        time.sleep(4)
    # Once a case simulation is completed,
    print "Case%d Completed" % caseIndex
    sys.stdout.flush()
    # begin monitoring the next case.
    caseIndex = caseIndex+1

import win32com
import win32com.client.dynamic
import os

def csvFilename(modelName, caseIndex):
    """Create the unique filename of the csv file."
    # Concatenate and return the unique csv filename.
    return modelName+' case '+str(caseIndex)+' .csv'

def exportCaseResults(app, modelIndex, modelName, caseIndex, outputDirectory):
    """Export the simulation results for the requested case to a unique csv file."
    # Create the filepath (including the filename) where the csv is to be saved.
    filepath = os.path.join(outputDirectory, csvFilename(modelName, caseIndex))
    print "%s" % filepath
    # Set the current case to the case from which the results are requested.
    app.View().SetCurrentCase(caseIndex)
    # Save the results to a variable (tables is a COM object, and can only be read by JMAG).
    tables = app.GetModel(modelIndex).GetStudy(0).GetResultTable()
    # Ensure the basic coordinate system for results.
    tables.SetXYZComponent(u"Absolute")
    # Write all results from the specified case to the unique filepath.

# and while the case hasn't finished
while (progress!=101):
    # check the progress of the simulation for that case
    progress = monitorSimulationProgress(modelName, caseIndex)
sys.stdout.flush()
    # and, if the simulation is in progress,
    if progress >=0:
        # wait between progress probes.
        time.sleep(4)
    # Once a case simulation is completed,
    print "Case%d Completed" % caseIndex
    sys.stdout.flush()
    # begin monitoring the next case.
    caseIndex = caseIndex+1

import win32com
import win32com.client.dynamic
import os

def csvFilename(modelName, caseIndex):
    """Create the unique filename of the csv file."
    # Concatenate and return the unique csv filename.
    return modelName+' case '+str(caseIndex)+' .csv'

def exportCaseResults(app, modelIndex, modelName, caseIndex, outputDirectory):
    """Export the simulation results for the requested case to a unique csv file."
    # Create the filepath (including the filename) where the csv is to be saved.
    filepath = os.path.join(outputDirectory, csvFilename(modelName, caseIndex))
    print "%s" % filepath
    # Set the current case to the case from which the results are requested.
    app.View().SetCurrentCase(caseIndex)
    # Save the results to a variable (tables is a COM object, and can only be read by JMAG).
    tables = app.GetModel(modelIndex).GetStudy(0).GetResultTable()
    # Ensure the basic coordinate system for results.
    tables.SetXYZComponent(u"Absolute")
    # Write all results from the specified case to the unique filepath.

# and while the case hasn't finished
while (progress!=101):
    # check the progress of the simulation for that case
    progress = monitorSimulationProgress(modelName, caseIndex)
sys.stdout.flush()
    # and, if the simulation is in progress,
    if progress >=0:
        # wait between progress probes.
        time.sleep(4)
    # Once a case simulation is completed,
    print "Case%d Completed" % caseIndex
    sys.stdout.flush()
    # begin monitoring the next case.
    caseIndex = caseIndex+1

import win32com
import win32com.client.dynamic
import os

def csvFilename(modelName, caseIndex):
    """Create the unique filename of the csv file."
    # Concatenate and return the unique csv filename.
    return modelName+' case '+str(caseIndex)+' .csv'

def exportCaseResults(app, modelIndex, modelName, caseIndex, outputDirectory):
    """Export the simulation results for the requested case to a unique csv file."
    # Create the filepath (including the filename) where the csv is to be saved.
    filepath = os.path.join(outputDirectory, csvFilename(modelName, caseIndex))
    print "%s" % filepath
    # Set the current case to the case from which the results are requested.
    app.View().SetCurrentCase(caseIndex)
    # Save the results to a variable (tables is a COM object, and can only be read by JMAG).
    tables = app.GetModel(modelIndex).GetStudy(0).GetResultTable()
    # Ensure the basic coordinate system for results.
    tables.SetXYZComponent(u"Absolute")
    # Write all results from the specified case to the unique filepath.
tables.WriteAllTables(filepath, u"Time")

return filepath

def exportSimulationResultsToCsv(app, modelIndex, modelName, outputDirectory):
    
    """Export the simulation results for each case to a unique csv file."
    
    # Check for new results for all cases. Simulations must be finished
    # before this line runs.
    
    # Otherwise, the simulation from which the results are requested
    # will end early.
    
    app.GetModel(modelIndex).GetStudy(0).CheckForNewResults()
    
    print "Exporting Simulation Results to CSV"
    
    # Set current study to the study from which results are desired.
    This line only works if you have ONLY ONE STUDY per model.
    
    app.setCurrentStudy(modelIndex)
    
    # Create the path to the folder where the results will be stored.
    #path = os.path.join('C:', os.sep, 'Users', 'crtaylor', 'Dropbox (MIT)
    #', 'motorCalculations', 'motorOptimizationCode', 'results')
    
    # Create empty filepath list for csv result files.
    filepathList = []
    
    # For each of the three cases.
    for caseIndex in [1, 2, 3]:
        
        # create a csv file with the results.
        filepathList.extend(exportCaseResults(app, modelIndex, modelName, caseIndex, outputDirectory))
    
    return filepathList

if __name__ == "__main__":
    
    app = win32com.client.dynamic.Dispatch("designer.Application")
    
    app.Show()
    exportSimulationResultsToCsv(app, 0, u"motorModel100")

import numpy as np
import csv
import pandas as pd
from pprint import pprint
import scipy.optimize
import scipy.linalg as la

def getPandasDataFrame(filepath, startIndex, separationSize, endIndex):
    rowskiplist = range(0, startIndex + 1)
    rowskiplist.extend(range(startIndex + separationSize, endIndex))
    data = pd.read_csv(filepath, skiprows=rowskiplist)
    
    # Find time column because JMAG isn't consistent in export
getColumn = [c for c in data.columns.values if "Time" in c][0]
data = data[data[getColumn]>0]
return data
def getResultsFromCsv(filepath):
    # Pull in all of the data from the csv file.
    with open(filepath, 'rb') as csvfile:
        reader = csv.reader(csvfile)
        data = list(reader)
    resultsIndexDict = {}
    for i in range(len(data)):
        line = data[i]
        checkLine = line[0].replace(" ","",1)
        if not checkLine.isdigit() and not 'STUDY-TITLE' in line[0].upper() and not 'Time' in line[0]:
            resultsIndexDict[line[0]]=i
    indexList = resultsIndexDict.values()
    indexList.sort()
    indexList = np.array(indexList)
    separationSizeList = list(indexList[1:]-indexList[:1])
    separationSize = separationSizeList[0]
    if separationSizeList.count(separationSize)!=len(separationSizeList):
        print "Woah! There is an inconsistency in number of steps in the results file."
        print filepath
        print separationSizeList
    resultsDict = {}
    for key in resultsIndexDict.keys():
        resultsDict[key] = getPandasDataFrame(filepath,resultsIndexDict[key],separationSize,len(data))
    return resultsDict
def preprocessResultsFromCsv(filepathList):
    preprocessedResults = dict((i,[[],[],[]]) for i in [
        'avgSecondaryIronLoss','avgCopperLoss','maxTorque','
        'minTorque','avgTorque','torqueRipple','maxVoltageAmplitude','
        maxCurrentAmplitude',
        'maxFluxLinkageAmplitude','motorVelocityRpm'])
    for i in range(len(filepathList)):
        resultsDict = getResultsFromCsv(filepathList[i])
        preprocessedResults['avgSecondaryIronLoss'][i] = resultsDict['
            Joule_Loss:W<Value>']>[][motorGeom3/FluxConcentrator^
            motorGeom3-1.3',
    return preprocessedResults
'motorGeom3/FluxConcentrator' 'motorGeom3-1.2',
'motorGeom3/FluxConcentrator' 'motorGeom3-1',
'motorGeom3/Magnet' 'motorGeom3-40',
'motorGeom3/Magnet' 'motorGeom3-1.2',
'motorGeom3/Magnet' 'motorGeom3-40.2',
'motorGeom3/Magnet' 'motorGeom3-1.3',
'motorGeom3/Magnet' 'motorGeom3-40.3',
'motorGeom3/Magnet' 'motorGeom3-1',
'motorGeom3/Return' 'motorGeom3-1'].sum(axis=1).mean()

preprocessedResults['avgCopperLoss'][i] = resultsDict['Joule Loss: W.<Value>'] ['motorGeom3/Coil' 'motorGeom3-1',
'motorGeom3/Coil' 'motorGeom3-2', 'motorGeom3/Coil' 'motorGeom3-3']
].sum(axis=1).mean()

preprocessedResults['maxTorque'][i] = resultsDict['Torque: Nm.<Value>'] ['First Phase Torque',
'Second Phase Torque', 'Third Phase Torque']
].sum(axis=1).max()

preprocessedResults['minTorque'][i] = resultsDict['Torque: Nm.<Value>'] ['First Phase Torque',
'Second Phase Torque', 'Third Phase Torque']
].sum(axis=1).min()

preprocessedResults['avgTorque'][i] = resultsDict['Torque: Nm.<Value>'] ['First Phase Torque',
'Second Phase Torque', 'Third Phase Torque']
].sum(axis=1).mean()

preprocessedResults['torqueRipple'][i] = (preprocessedResults['maxTorque'][i] - preprocessedResults['minTorque'][i]) /
preprocessedResults['avgTorque'][i]

preprocessedResults['maxVoltageAmplitude'][i] = resultsDict['Circuit Voltage: V.<Value>'] ['VP1', 'VP2', 'VP3']
.abs().max().max()

preprocessedResults['maxCurrentAmplitude'][i] = resultsDict['Current: A.<Value>'] ['Coil1', 'Coil2', 'Coil3']
.abs().max().mean()

preprocessedResults['maxFluxLinkageAmplitude'][i] = resultsDict['FEM_Coil_Flux-Linkage: Wb.<Value>'] ['Coil1', 'Coil2', 'Coil3']
.abs().max().mean()

preprocessedResults['motorVelocityRpm'][i] = resultsDict['Rotational Velocity: r/min.<Value>'] ['Motor Velocity']
.mean()

return preprocessedResults

def curveFit(X,Y):

\[
\text{Beta} = \text{np.dot}(\text{np.dot}(\text{la.inv}(\text{np.dot}(X.T,X)),X.T),Y)
\]
\[
\text{sumSquaredError} = \text{la.norm}(\text{np.dot}(X, \text{Beta})-Y)\ast 2
\]
return Beta, sumSquaredError

def calculateCopperDissipationConstant(preprocessedResults):
    \[
    \text{tauSquared} = \text{np.array([preprocessedResults['avgTorque']])}.T\ast 2
    \]
    \[
    \text{pCopper} = \text{np.array([preprocessedResults['avgCopperLoss']])}.T
    \]
    \[
    \text{kCopper}, \text{sse} = \text{curveFit}(\text{tauSquared}, \text{pCopper})
    \]
    # Return the optimal scalar
    return kCopper[0,0]

def exponentialObjectiveFunction(k, motorParams=(0,0)):
    '''Objective function to optimize for Iron Loss Constants'''
    \[
    \text{f} = \text{motorParams[0]}
    \]
    \[
    \text{pIron2} = \text{motorParams[1]}
    \]
    \[
    \text{cost} = 0
    \]
    for \text{i in range(len(f))}:
        \[
        \text{cost} += (\text{pIron2}[i]-(k[0]*f[i]**k[1]))\ast 2
        \]
    return cost

def calculateSecondaryIronLossConstants(preprocessedResults):
    \[
    \text{n} = \text{np.array([preprocessedResults['motorVelocityRpm']])}.T
    \]
    \[
    \text{p} = 20.0
    \]
    \[
    \text{f} = \text{n*p/60.0}
    \]
    \[
    \text{pIron2} = \text{np.array([preprocessedResults['avgSecondaryIronLoss']])}.T
    \]
    \[
    \text{res} = \text{scipy.optimize.minimize(\text{exponentialObjectiveFunction},[0.002, 1.6], \text{args}=(\text{f}, \text{pIron2}), \text{method}="\text{nelder-meand}, \text{options}={"\text{disp}: \text{False}}\}}
    \]
    # Return the optimal parameters (k,a) for pIron2 = k*f^a
    return res.x

def logarithmicObjectiveFunction(x, motorParams=(9,18,27),(0,0,0)):
    '''Objective function to optimize for torque constant
    \text{Note: x[0] is K.Tt and x[1] is a.kt}'''
    \[
    \text{I} = \text{motorParams[0]}
    \]
    \[
    \text{T} = \text{motorParams[1]}
    \]
    \[
    \text{cost} = 0
    \]
    for \text{i in range(len(I))}:
        \[
        \text{cost} += (T[i]-x[0]/x[1]*\text{np.log}(x[1]*I[i]+1))\ast 2
        \]
    return cost

def calculateImax(torqueList, maxReqdTorque, l=(9,18,27)):
    # Calculate torque constant
    \[
    \text{x0} = [1.0, 1.0]
    \]
    \[
    \text{res} = \text{scipy.optimize.minimize(\text{logarithmicObjectiveFunction}, x0 = x0, args=(l, torqueList), \text{method}="\text{nelder-meand}, \text{options}={"\text{disp}: \text{False}}\}}
    \]
k, a = res.x

# Calculate max Current
Imax = (np.exp(a*maxReqdTorque/k) - 1.0)/a

# threePhaseTorqueConstant = (11*torqueList[0] + 12*torqueList[1] + 13*torqueList[2])/float(11**2 + 12**2 + 13**2)

# Imax = maxReqdTorque/threePhaseTorqueConstant
return Imax, k, a

def processCsvSimulationResults(filepathList):
    preprocessedResults = preprocessResultsFromCsv(filepathList)
    maxVoltageAmplitude = max(preprocessedResults['maxVoltageAmplitude'])
    kCopper = calculateCopperDissipationConstant(preprocessedResults)
    kIron2, aIron2 = calculateSecondaryIronLossConstants(preprocessedResults)

    processedResultsDict = {
        "maxVoltageAmplitude": maxVoltageAmplitude,
        kCopper: kCopper,
        kIron2: kIron2,
        aIron2: aIron2
    }
    return processedResultsDict

# calculatePoleIronLossConstants.py
import numpy as np
import scipy.optimize

def exponentialObjectiveFunction(k, motorParams=(0, 0)):
    '''Objective function to optimize for Iron Loss Constants'''
    f = motorParams[0]
    pIron2 = motorParams[1]
    cost = 0
    for i in range(len(f)):
        cost += (pIron2[i] - (k[0]*f[i]*k[1]))**2
    return cost

def calculateIronLossConstants(f, pIron):
    res = scipy.optimize.minimize(exponentialObjectiveFunction,
                                  [0.002, 1.6], args=(f, pIron), method="nelder-mead", options={"disp": False})
    # Return the optimal parameters (k, a) for pIron2 = k*f^a
    return res.x

def calculatePoleIronLossConstants(features):
    f = np.array([0.66, 132, 198])
    pIron = np.array([0.1, 912800921, 5.106833333, 9.066731367])
    pIronRef = pIron[3]
    pIron = 2.0*pIron  # Multiply by 2 to account for the four sets of poles in 3 phases, vs two sets of poles in 1 phase.
    kIron, aIron = calculateIronLossConstants(f, pIron)
    rpo = np.array([28, 29, 30.125, 30.375])
pIronRpo = np.array
    ([7.934905961, 7.914657248, 9.066731367, 8.95541128])

rpoVal = features['rpo']
plironRpoInterp = np.interp(rpoVal, rpo, pIronRpo)
percentChangeDueToRpo = (plironRpoInterp - plironRef) / plironRef

nl = np.array([5, 6, 7, 8, 9])
tl = 0.2
ts = 0.05
wp = 2 * nl * tl + (2 * nl - 1) * ts
plironWp = np.array
    ([6.162728047, 7.956413343, 9.066731367, 9.990574073, 10.00716176])

wpVal = features['wp']
plironWpInterp = np.interp(wpVal, wp, plironWp)
percentChangeDueToWp = (plironWpInterp - plironRef) / plironRef
kIronNew = kIron * (1.0 + percentChangeDueToRpo + percentChangeDueToWp)

poleIronLossResults = {'kIron': kIronNew, 'aIron': aIron}
return poleIronLossResults

import scipy.io as sio
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib as mpl
import numpy as np
def deriv1(x, h):
    ...
    Take the derivative using a five point stencil, accounting for boundaries.
    (1st)  -25  48  -36  16  -3
    (2nd)  -3  -10  18  -6   1
    (Middle)  1   -8   0   8   -1      / 12h
    (Penult.)  -1   6  -18  10   3
    (Last)  3   -16  36  -48  25
    The input vector is x (which must be a least five elements in length).
    and the time step is h.
    ...
    xp = np.zeros(len(x))
    for i in range(len(x)):
        if i == 0:
        elif i == 1:

elif i == (len(x) - 2):
    xp[i] = (-1*x[i-3] + 6*x[i-2] - 18*x[i-1] + 10*x[i] + 3*x[i+1])/(12*h)

elif i == (len(x) - 1):
    xp[i] = (3*x[i-4] - 16*x[i-3] + 36*x[i-2] - 48*x[i-1] + 25*x[i])/(12*h)

else:
    xp[i] = (1*x[i-2] - 8*x[i-1] + 6*x[i] + 8*x[i+1] - 1*x[i+2])/(12*h)

return xp

def loadAndTransformTorqueSpeedCurve(features):
    mat = sio.loadmat('C:\Users\crtaylor\Dropbox-(MIT)\biomData\FastStep_2681b_User_step57')
    ...

    for reference, info = sio.whosmat('C:\Users\crtaylor\Dropbox (MIT)\biomData\FastStep_2681b_User_step57')
    displays names, previous shapes, and types of all matrices in the mat file.
    ...

    # Motor Torque
    maxonTorque = mat['motor_Torque_Nm']

    # Timestep
    stepSize = 2e-3

    # Angular Velocity (RPM)
    maxonVelocityRpm = -mat['motor_AngularVelocity_RPM']

    # Angular Velocity (RAD/s)
    maxonVelocityRadPerSec = maxonVelocityRpm * 2 * np.pi / 60

    # Angular Acceleration (RAD/s**2)
    maxonAccelRadPerSec2 = deriv(maxonVelocityRadPerSec, stepSize)

    # Subtract out the Maxon rotor inertia to get the actual required torque at the previous tx ratio.
    maxonRotorInertia = 33.3/1e7 # Change this to zero if you wish to temporarily remove the effect.
    maxonAccelRadPerSec2 = deriv(maxonVelocityRadPerSec, stepSize)
    externalTorque = maxonTorque - maxonRotorInertia * maxonAccelRadPerSec2

    # Get the required torque and velocity after the transmission ratio alteration.
    transformedTorque = externalTorque * 11
    requiredVelocityRadPerSec = maxonVelocityRadPerSec / 11
    requiredAccelRadPerSec2 = maxonAccelRadPerSec2 / 11
requiredVelocityRpm = maxonVelocityRpm/11
# Get the total required torque after accounting for the additional rotor inertia.
newMotorInertia = features["Iz"] # kg-m^2 (Baseline Transverse Flux Motor Inertia)
requiredTorque = transformedTorque + newMotorInertia*
requiredAccelRadPerSec2
return requiredVelocityRpm, requiredTorque, requiredVelocityRadPerSec

def calculateAveragePowerLoss( features , processedResults ,
poleIronLossResults , returnExtra=False):
    requiredVelocityRpm , requiredTorque , requiredVelocityRadPerSec =
    loadAndTransformTorqueSpeedCurve( features )
p = 20.0
requiredFrequencies = np.abs(requiredVelocityRpm*p/60.0)
pcopper = processedResults["kCopper"]*requiredTorque**2.0
plron2 = processedResults["kIron2"]*requiredFrequencies**
processedResults["kIron"]
plron = poleIronLossResults["kIron"]*requiredFrequencies**
poleIronLossResults["kIron"]
ploss = pcopper + plron2 + plron
pout = requiredTorque*requiredVelocityRadPerSec
averagePositivePowerOut = np.mean(np.clip(pOut ,0 ,np.max(pOut)))
averagePowerOut = np.mean(pOut)
averageCopperLoss = np.mean(pcopper)
averagePoleIronLoss = np.mean(plron)
averageSecondaryIronLoss = np.mean(plron2)
averagePowerLoss = np.mean(pLoss)
averagePowerIn = averagePowerOut + averagePowerLoss
netPowerEfficiency = averagePowerOut/(averagePowerOut +
    averageCopperLoss + averagePoleIronLoss +
    averageSecondaryIronLoss)
positivePowerEfficiency = averagePositivePowerOut/(averagePositivePowerOut +
    averageCopperLoss + averagePoleIronLoss +
    averageSecondaryIronLoss)
print 'Copper-Loss: ', averageCopperLoss
print 'Pole-Iron -Loss: ', averagePoleIronLoss
print 'Total-Loss: ', averagePowerLoss
print 'Power-Out: ', averagePowerOut
print "Average -Power-n:  ", averagePowerIn
if returnExtra:
    return averagePowerLoss , netPowerEfficiency ,
positivePowerEfficiency, max(abs(requiredTorque))
return averagePowerLoss
Bibliography


