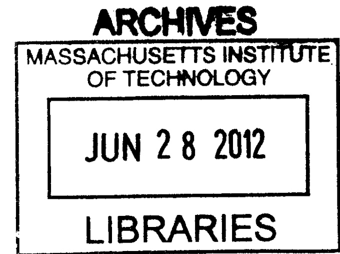
Simulating Control of the Ankle Joint

by

Rebecca Vasquez


ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 28 2012
LIBRARIES

Submitted to the
Department of Mechanical Engineering
in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Mechanical Engineering

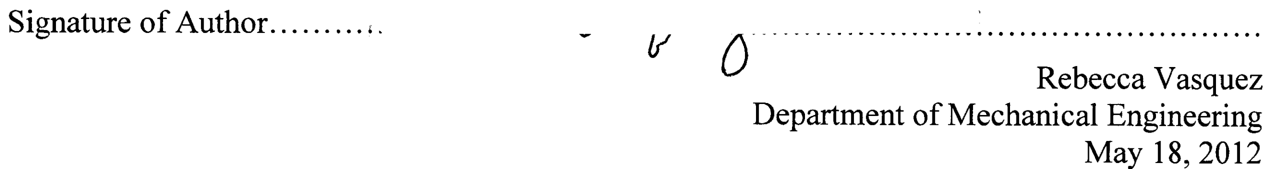
at the

Massachusetts Institute of Technology

June 2012

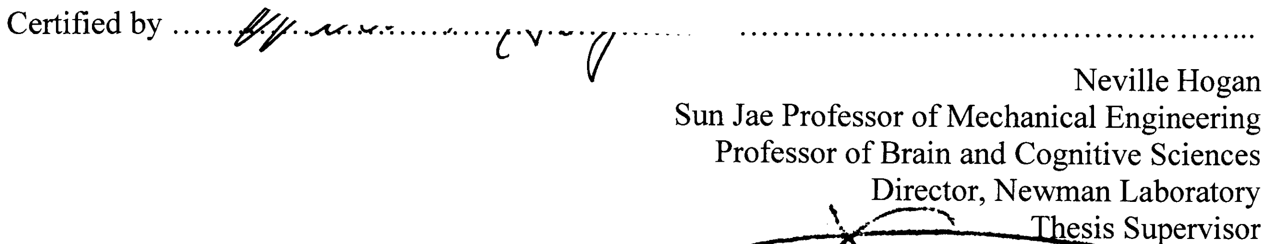
© 2012 Rebecca Vasquez
All rights reserved

The author hereby grants to MIT permission to reproduce and to
distribute publicly paper and electronic copies of this thesis document in whole or in part in any
medium now known or hereafter created.



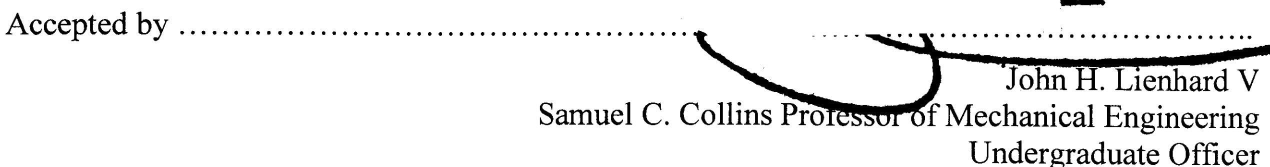
Signature of Author..

Rebecca Vasquez
Department of Mechanical Engineering
May 18, 2012



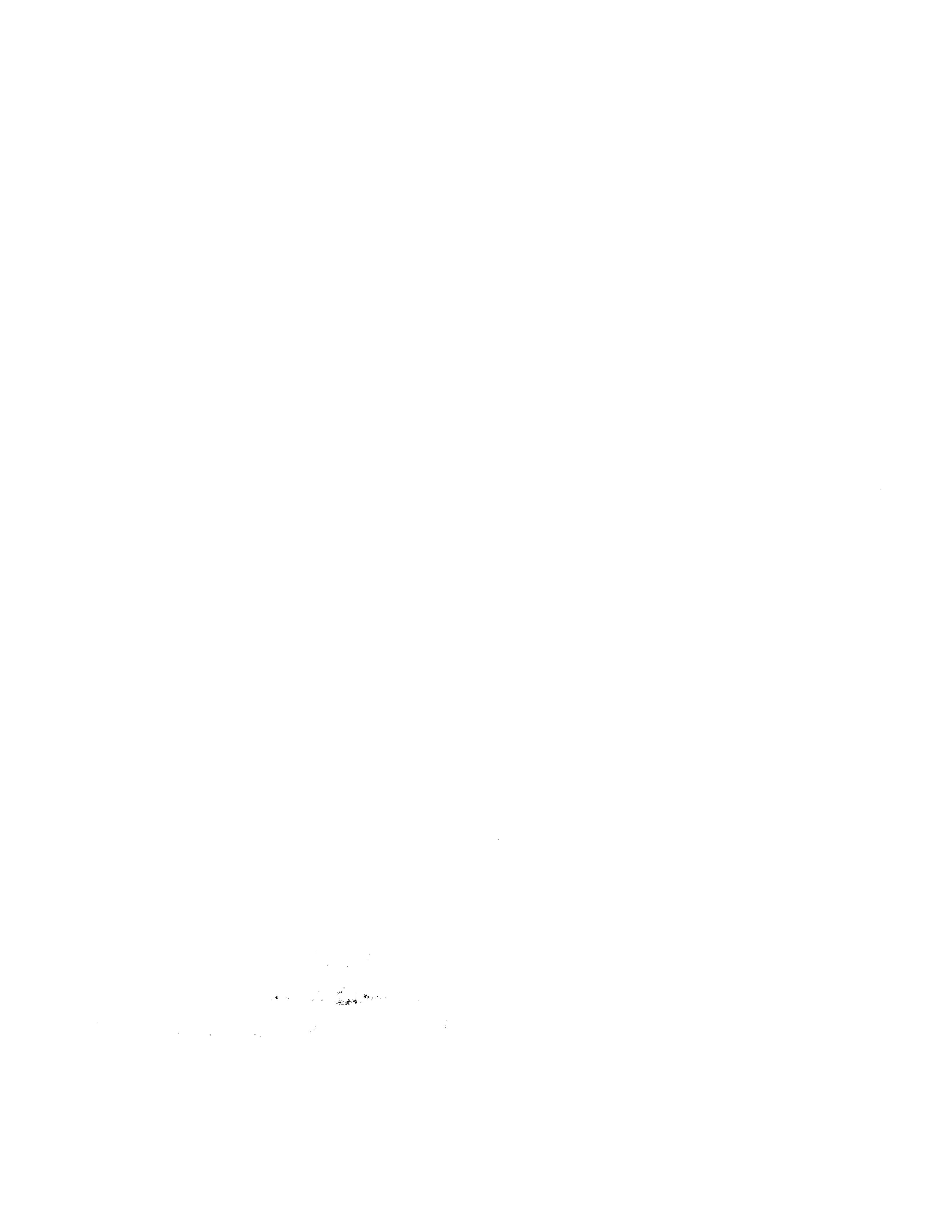
Certified by ....

Neville Hogan
Sun Jae Professor of Mechanical Engineering
Professor of Brain and Cognitive Sciences
Director, Newman Laboratory
Thesis Supervisor



Accepted by ....

John H. Lienhard V
Samuel C. Collins Professor of Mechanical Engineering
Undergraduate Officer



Simulating Control of the Ankle Joint

By: Rebecca Vasquez

Submitted to the Department of Mechanical Engineering
On May 18, 2012 in Partial Fulfillment of the
Requirements for the Degree of Bachelor of Science in
Mechanical Engineering

ABSTRACT

Computing environments such as Matlab that are conventionally used to simulate dynamics of rigid body systems can be used to model interactions between the system and its environment. However, creating these simulations using Matlab or an equivalent is difficult and there is a need for a more convenient simulation environment for such problems. Two alternative programs, PyODE and OpenSim, were explored to evaluate their ability to fill this need. Models and simulations of the human ankle were created in PyODE. This program is useful for creating simple models where the programmer desires a high level of control over model parameters. Simulations of the ankle kicking a ball and taking a step were created to examine the effect of joint stiffness on these motions and help determine the usefulness of ODE as a simulation tool. Pre-existing models were analyzed in OpenSim. OpenSim is specifically designed for analyzing biomechanical systems. It allows for more complex models to be created but the user has more limited control over the model parameters.

Thesis Supervisor: Neville Hogan

Title: Sun Jae Professor of Mechanical Engineering, Professor of Brain and Cognitive Sciences, Director, Newman Laboratory

ACKNOWLEDGEMENTS

I would like to thank my adviser, Professor Neville Hogan, for giving me the opportunity to work with him and his students to complete this project. His guidance, expertise, and feedback have been tremendously helpful.

I would also like to thank Will Bosworth for his time, assistance, advice, and support throughout this project. I could not have accomplished this without his help.

Finally I would like to thank my family and friends for supporting me and offering aid wherever they could.

TABLE OF CONTENTS

ABSTRACT 3
TABLE OF CONTENTS 5
TABLE OF FIGURES 6
1. INTRODUCTION 7
2. SIMULATION SOFTWARE 9
2.1 Open Dynamics Engine 9
2.2 OpenSim 10
3.1 Actuators Coaxial with the Joint 13
3.1.1 Kinematics 13
3.1.2 Model in PyODE 13
3.1.3 Muscle Control 14
3.2 Linear Actuators to Simulate Muscle 15
3.2.1 Kinematics 15
3.2.2 Model in PyODE 16
3.2.3 Muscle Control 17
4.1 Joint Angle Control in PyODE 19
4.2 Kicking a Ball in PyODE 21
4.3 Simulating Stepping 25
4.4 Model Analysis in OpenSim 28
5. DISCUSSION 30
5.1 Comparing ODE and OpenSim 30
5.2 Potential Significance 31
5.3 Future Investigation 31
Appendix A: PyODE Code for Simulations 34
Appendix B: Links to installation files 39

TABLE OF FIGURES

Figure 1: OpenSim model of two legs. Front view (left) and side view (right). 10
Figure 2: OpenSim model of two legs with walking motion loaded, legs in mid stride. 12
Figure 3: PyODE model of ankle. Actuator located at and coaxial with the ankle joint created 1 degree of freedom rotation about the z axis (dorsi-plantar flexion) 13
Figure 4: Proportional plus rate control feedback loop for 1 DOF ankle model with actuator at the joint. 15
Figure 5: PyODE model of an ankle with increased complexity. Linear actuators connect the shin and foot to simulate muscles and allow for rotation about the z axis. 17
Figure 6: Response of models to changing ankle reference angle. (a) and (b) show the angle and joint torques for the actuator at joint model; (c) and (d) show the angle and joint torques for the linear actuator model.. 19
Figure 7: Magnitude of ankle deflection (rad) after the application of an external force to the foot in (a) the model with coaxial actuators and (b) the model with linear actuators. 20
Figure 8: Linear muscle model with moment arms shown. The moment arm is used to approximate joint stiffness. 21
Figure 9: Ankle model with ball. The 'ball' is a cylinder with its axis in the z direction. 22
Figure 10: Final ball position as a function of stiffness for (a) actuator at the joint and (b) linear actuators. Ball started at x = 0.3048 m.. 23
Figure 11: Contact point between foot and ball changes when ball starts in a different location. Ball at 0.3048 m (left) vs. ball at 0.381m (right) 24
Figure 12: Final ball position as a function of stiffness for (a) actuator at the joint and (b) linear actuators. Ball started at x = 0.381 m... 24
Figure 13: Progression of a drop test. In frame 1 the shin is stationary. Then the fixed joint is removed and the whole ankle falls. The foot makes contact with the ground (dotted line) in frame 3. 25
Figure 14: Response of models to drop test. (a) and (b) show ankle angle and joint torque for actuator at the joint model, (c) and (d) show ankle angle and joint torque for linear actuator model. The ankle angle is measured between the foot and shin. The shin is free to rotate during the drop. 26
Figure 15: Response of models to drop test. (a) & (b) knee trajectory for actuator at the joint model, (c) and (d) knee trajectory for linear actuator model. Starting foot position and joint/muscle stiffness are varied.. 27
Figure 16: Plot of muscle force during walking motion for peroneus longus muscle. In (a) the maximum isometric force was set to 756 N and in (b) the maximum isometric force was set to 500 N.. 29

1. INTRODUCTION

Computer simulations are important tools to use for understanding the motion and control of biomechanical systems. Computing programs such as Matlab can be used to model the dynamic behavior of rigid body systems. The responses of these systems can be analyzed using differential equation solvers. A disadvantage of Matlab as a computing environment for these dynamic simulations is that there are few tools available to handle complex problems such as collision detection or constraint satisfaction. Another disadvantage of Matlab as a simulation environment is that it is difficult to produce visual representations of simulations. It is often useful and interesting to see a visual representation of the system behavior in the form of an animation as the simulation is running to connect the analytic response to the physical motion.

Two programs have been identified that may provide alternative simulation environments, each with animation capabilities. One is Open Dynamics Engine, a code library used to simulate rigid body dynamics. Open Dynamics Engine, or ODE, is conventionally used as a tool to create video games and simulations [1]. The other is OpenSim, created by SimTK. OpenSim is specifically designed for creating and running simulations on musculoskeletal models of humans and animals [2].

The goal of this study was to evaluate each program in terms of its capability to create models of biomechanical systems with various levels of complexity and run simulations on them. In order to accomplish this, a model of a human ankle was created using ODE and a muscle controller was implemented to simulate neural control of the movement of the ankle. Muscles were modeled as torque motors located coaxial with the ankle joint. A second model was created in ODE, increasing the complexity of the system by modeling muscles as linear actuators connecting the shin and foot. Simulations of motions requiring only one degree of freedom at

the ankle were run with each model. One motion was kicking a ball and another was taking a step. These simulations were each run with varying joint stiffness to examine the effect of stiffness on the resulting motion. The goals were to find an ideal ankle stiffness that would make the ball travel farthest following a kick and determine whether ankle stiffness had an effect on the motion of the knee during stepping.

When OpenSim was downloaded, example models of anatomically correct musculoskeletal structures were included. The existence of these models showed that it is possible to create a model of the ankle in OpenSim with a level of anatomical detail and accuracy at least as great as that possible with ODE and makes further model construction superfluous. Instead, the simulation capabilities of OpenSim were explored to determine what types of analysis the program is best suited for.

2. SIMULATION SOFTWARE

2.1 Open Dynamics Engine

Open Dynamics Engine (ODE) is a code library used to simulate rigid body dynamics. ODE is written in C/C++ [1]. This study used PyODE which has the same functionality as ODE but the default language is Python.

To create a simulation in PyODE, a world object must first be created to hold all of the rigid bodies. Once this world exists, bodies can be created by using instances of the body class. Bodies are defined by their mass properties: total mass, center of gravity, and inertia tensor. For simple geometries such as a box, cylinder, or sphere these mass properties can be set by defining the shape and then giving the density of the body and key dimensions such as length or radius. Forces can be applied to bodies using existing functions in PyODE. There are also functions that will return the position of the body in space and any forces that are acting on the body.

Bodies are connected to each other and to the environment by joints. PyODE has ten different joint types. When setting a joint, the anchor point and the axis/axes of rotation/translation are defined. There are built in functions for applying torques or forces at joints and certain joint types can be set to move at a defined velocity. There are also functions for getting feedback from a joint which includes the torques and forces on the joint as well as the position of the joint.

There is built in collision detection in PyODE and a special joint type called a contact joint is used to tell which bodies have collided at any given time. This will also prevent two colliding bodies from moving through one another. In order for collision detection to take place each body must have a geometry object associated with it. Though collision detection functions

can be written in Matlab, none are available as a part of the software and it is not trivial to create such programs.

Once all the bodies and joints for a simulation have been created the physics engine within PyODE will update the state of the world object at specified time intervals. Control systems can be implemented within the simulation and feedback can be gathered from the model at each time step to track position, joint angles and velocities, and other quantities of interest. The display (created with pyOpenGL) was set to update at each time step to match the new state of the simulated world, creating an animation.

2.2 OpenSim

As mentioned previously, OpenSim comes with example models that can be used to run simulations. The model that is of interest in this study is an anatomically detailed model of the musculoskeletal structure of human legs. This model is shown in Figure 1.

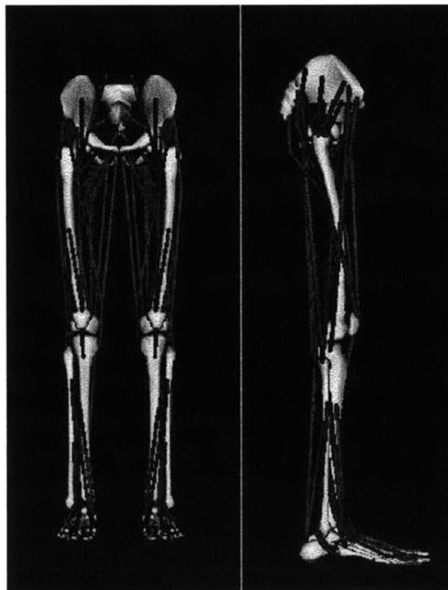


Figure 1: OpenSim model of two legs. Front view (left) and side view (right).

To develop new models in OpenSim the language used is C++. SimTK provides a user guide with an introduction to creating models for use in OpenSim. Similar to PyODE, bodies are created and connected to each other and the environment by joints. The specific language used to define these processes is different but the results are essentially the same. There are also functions for setting gravity, adding forces, collisions, and getting feedback from the model analogous to those found in PyODE.

OpenSim was created for the specific function of modeling biomechanical systems and it has additional functionality built in to facilitate this. One such feature is the ability to create and simulate muscles using the actuator subclass called 'Muscle'. Muscles are defined by their maximum isometric force, optimal fiber length, tendon slack length, pennation angle, and activation/deactivation time constants. The developer also sets the initial activation and fiber length for each muscle in the model [3]. There is a built in controller class that enables the creation of different muscle controllers. It can be used to implement a PD controller that allows for muscle mechanical impedance to be defined by setting stiffness and damping coefficients for the muscle. OpenSim includes other actuator classes in addition to the muscle class that can be used when creating models; custom actuators can be defined as well.

To visualize simulations on models in the OpenSim user interface a motion file needs to be written in addition to the model file. When a model is written must include an integrator to handle the forward dynamics simulation. Integrators are SimTK objects that simulate the behavior of a system over time. Once the system of equations for the model has been integrated, the resulting motion can be saved as a motion file [3]. This is then loaded into the user interface along with the model to generate an animation of the simulation. There are pre-existing motions

associated with the models that come with the OpenSim download. One motion for the legs is walking; Figure 2 shows a picture of the model in mid stride.

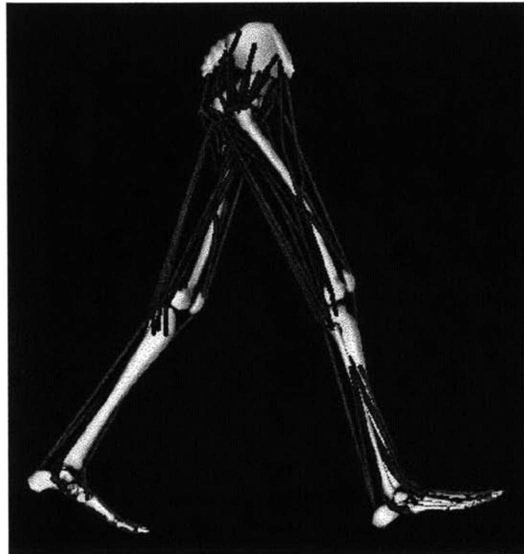


Figure 2: OpenSim model of two legs with walking motion loaded, legs in mid stride.

3. MODEL FORMATION IN ODE

3.1 Actuators Coaxial with the Joint

3.1.1 Kinematics

The first model approximated the shin and foot as cylinders. The foot was connected to the shin at the base of the shin such that both a heel and a toe extended from the joint. The foot had one degree of freedom relative to the shin (dorsi-plantar flexion): it could rotate around the z-axis (see Figure 3). The toe position depends on the length of the foot and the joint angle between the foot and the shin. This angle was controlled by a controllable-torque actuator located directly at and coaxial with the ankle joint.

3.1.2 Model in PyODE

The lower leg was modeled as a vertical cylinder held in space by creating a fixed joint between the cylinder and the environment. The foot was another cylinder, attached to the shin by a hinge joint that allowed rotation about the z axis. Figure 3 shows a picture of the model.

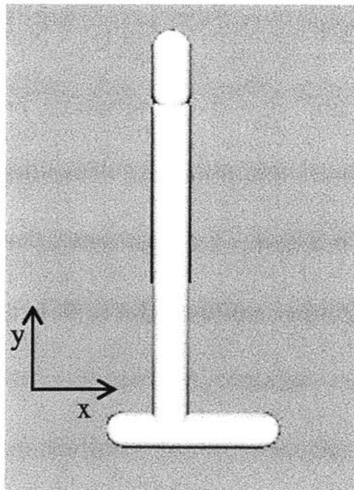


Figure 3: PyODE model of ankle. Actuator located at and coaxial with the ankle joint created 1 degree of freedom rotation about the z axis (dorsi-plantar flexion)

The angle of rotation around the z-axis (dorsi-plantar flexion) is the state variable of the model. Stops were added using the ODE method joint.setParam(HiStop/LoStop) to prevent the joint from moving beyond a reasonable range of motion for a human ankle. The range of motion was obtained from observation of ankles and extended from $\pi/6$ to $-\pi/3$ radians for dorsi-plantar flexion where zero was defined as the foot perpendicular to the shin. The actuator that controlled this movement was located directly at and coaxial with the joint and the torque it provided was controlled using the method 'addTorque' of the hinge joint class.

3.1.3 Muscle Control

The dynamics of the ankle joint were approximated as a linear spring and damper by assuming proportional plus derivative control of the joint. The amount of torque applied by the motor at the joint was a function of the joint angle and the rate of change of the joint angle. The stiffness and damping of the system determined the mechanical impedance of the ankle motor and were determined by the proportional and rate feedback gains respectively. The stiffness of the human ankle in dorsi-plantar flexion ranges from 9 to 34 N-m/rad depending on whether the foot is in dorsiflexion (~9 N-m/rad) or plantar flexion (~34 N-m/rad) [4]. This was implemented by measuring the joint angle and applying the appropriate proportional feedback gain, K, for each time step in the simulation. Ankle damping is in the range of 0.02 to 0.8 N-m-s/rad; the rate feedback gain, B, which implemented damping, was set to 0.3 N-m-s/rad for the simulations in this study [5]. The feedback loop for this ankle model is shown in Figure 4.

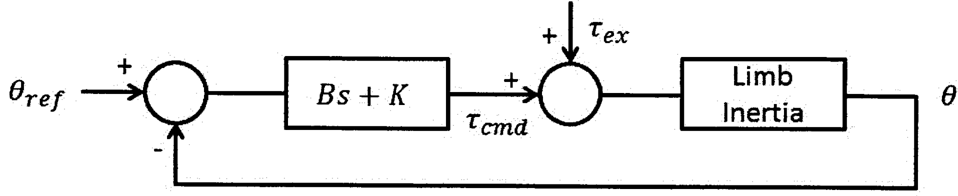


Figure 4: Proportional plus rate control feedback loop for 1 DOF ankle model with actuator at the joint.

Here θ_{ref} is the desired joint angle, θ is the actual joint angle, K is the joint stiffness in N-m/rad, and B is the joint damping in N-m-s/rad. The effects of gravity, inertia, and any other applied forces were generated by the physics engine within ODE. The call 'world.step(dt)' in the simulation code caused the model to update by a time step of size *dt*. The command torque sent to the muscle was defined as

$$\tau_{cmd} = K(\theta_{ref} - \theta) + B(\dot{\theta}_{ref} - \dot{\theta}) \tag{1}$$

The parameters θ and $\dot{\theta}$ were obtained using the 'getfeedback' method of the joint class in PyODE to determine the joint angle and the rate of change of the joint angle at each time step. The command torque was calculated and re-set at each time step to maintain the desired joint angle.

3.2 Linear Actuators to Simulate Muscle

3.2.1 Kinematics

In this second model, the ankle was represented again as two cylinders. However, in this case they were connected by linear joints that allowed for motion linearly along a defined axis. The linear actuators simulated muscles and replaced the motors located coaxial with the joint

from the previous simulation. The toe position depended on the joint angle and the angle also determined the lengths of the muscles.

3.2.2 Model in PyODE

This model was made using the previous model as a framework and changing the actuators at the joint to linear actuators which simulated muscle. The linear actuators were created using the slider joint type in PyODE. This joint allows two bodies to move linearly with respect to each other along an axis. The distance between the two bodies connected by a slider joint can be tracked to determine the length of the simulated muscle and forces can be added along the axis of the slider to change the length of the joint.

To appropriately simulate the movement of the joint with this type of actuator, the axis of translation needed to change as the foot rotated about the shin. To accomplish this additional bodies were needed in the model to attach the linear actuators to the shin and foot. These attachment bodies were small cylinders connected to the shin and foot at appropriate locations using a hinge joint that was free to rotate around the z-axis. The attachment bodies on the shin were then connected to their corresponding body on the foot by a slider joint. Figure 5 shows a picture of this model.

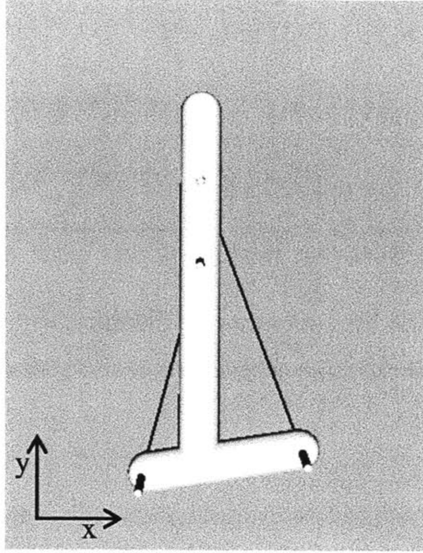


Figure 5: PyODE model of an ankle with increased complexity. Linear actuators connect the shin and foot to simulate muscles and allow for rotation about the z axis.

3.2.3 Muscle Control

The force applied by the linear actuator muscle was a function of the length of the muscle and the rate of change of length of the muscle. Once again the physics engine updated the world at each time step and a command force was sent to each muscle. The command force was defined as

$$F_{cmd} = k(\Delta l_{ref} - \Delta l) - b(v) \tag{2}$$

where v is the rate of change of length of the muscle, k is the muscle stiffness in N-m, and b is the muscle damping in N-m-s, Δl_{ref} is the necessary change in length from rest due to reach defined joint angle (defined reference angle assumed to be constant in this case), and Δl is the actual change in length. Specifically they are defined as

$$\Delta l_{ref} = l_0 - l_{ref} \tag{3}$$

$$\Delta l = l_0 - l \tag{4}$$

where l_0 is the rest length of the muscle when the joint angle is 0, l_{ref} is the length of the muscle when the joint angle is equal to the desired reference angle, and l is the actual length of the muscle. The code was written such that positive forces push against the foot (lengthen the muscle) and negative forces pull on the foot (shorten the muscle).

4. RUNNING SIMULATIONS

4.1 Joint Angle Control in PyODE

In ODE a basic level of control over the ankle model was achieved by defining a reference angle. The plots in Figure 6 show the response of the models to changing reference angles. For $0 \leq t < 0.8$ seconds the reference angle was set to 0 radians, for $0.8 \leq t < 1.6$ seconds the reference angle was set to $\pi/6$ radians, and for $1.6 \leq t \leq 2.4$ seconds the reference angle was set to - $\pi/6$ radians. In all simulations in this study the time step was 0.008 seconds.

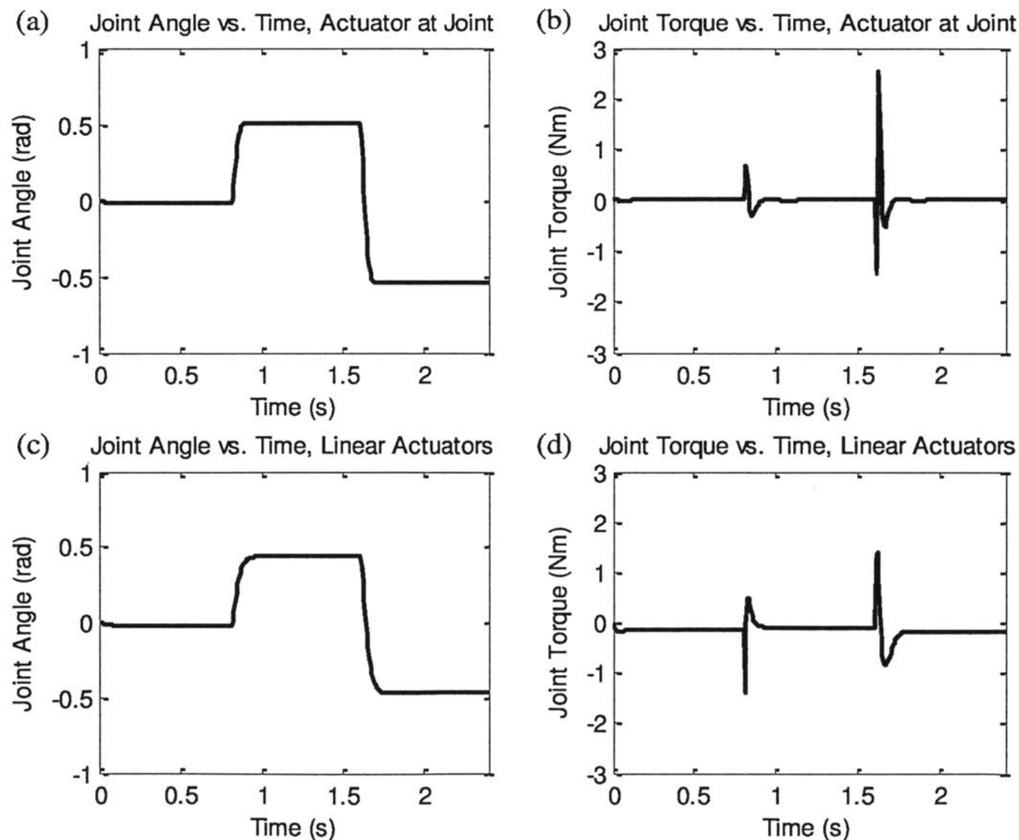


Figure 6: Response of models to changing ankle reference angle. (a) and (b) show the angle and joint torques for the actuator at joint model; (c) and (d) show the angle and joint torques for the linear actuator model.

In the linear actuator model the reference angle was changed by changing the lengths of the muscles. In figures (a) and (b) the second transition is more oscillatory than the first. This was due to the difference in joint stiffness between plantar flexion and dorsiflexion.

External forces were added to this simulation to analyze the response of the ankle to an impulse. This simulation was run on the model with an actuator coaxial with the joint with varying external force magnitude and varying K values to determine the relationship between maximum angular deflection and joint stiffness. The results are shown in Figure 7. As expected, foot deflection increased with decreasing joint stiffness and with increasing force for both models.

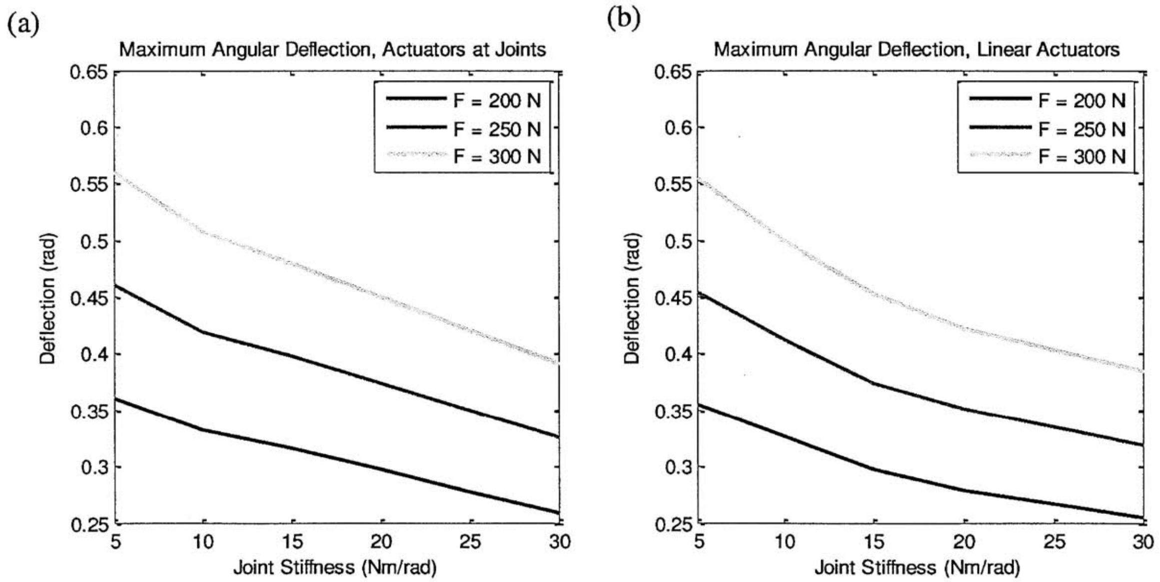


Figure 7: Magnitude of ankle deflection (rad) after the application of an external force to the foot in (a) the model with coaxial actuators and (b) the model with linear actuators.

In all simulations in this study the joint stiffness for the coaxial actuator model was set directly by defining a value for K for the ankle during the simulation. The stiffness for the linear actuator model was set by approximating joint stiffness as

$$K = k_1 r_1^2 + k_2 r_2^2 \tag{5}$$

where k_1 and k_2 are the individual muscle stiffness values and r_1 and r_2 are the moment arms of the muscles shown in Figure 8.

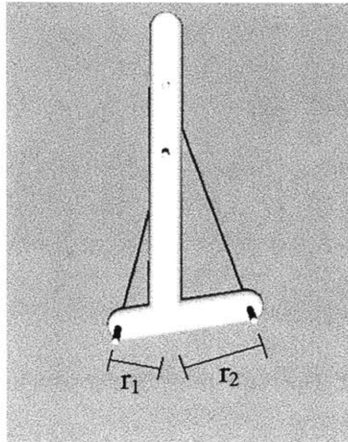


Figure 8: Linear muscle model with moment arms shown. The moment arm is used to approximate joint stiffness.

Originally a constant muscle stiffness value was set that would result in a desired joint stiffness when the ankle angle was equal to the defined reference angle of 0 radians. As the ankle moved, the muscle stiffness values remained constant despite changing moment arm lengths, which resulted in slight variation of joint stiffness during the simulation. The simulation was recreated such that the moment arms were calculated and the muscle stiffness values were redefined at each time step to eliminate this variation.

4.2 Kicking a Ball in PyODE

To simulate kicking a ball, the original models were changed so that the knee joint was hinged rather than fixed with respect to the environment. This allowed the shin to rotate around a z axis through the knee and simulate a kicking motion. A new body was introduced to the simulation environment to represent a ball. It was created as a cylinder whose axis was in the z direction so that it could roll in the direction of the kick. A planar surface was included in this model to provide a floor for the ball to roll on. A picture of the new model is shown in Figure 9.

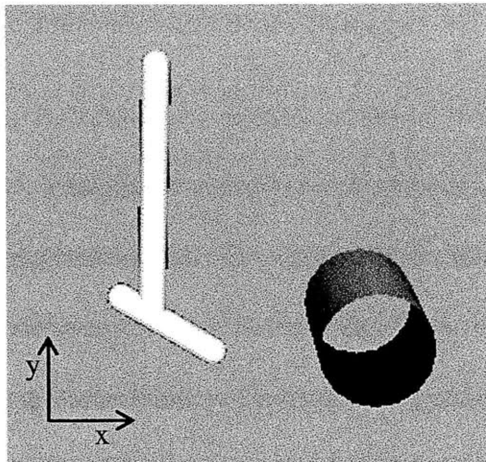


Figure 9: Ankle model with ball. The 'ball' is a cylinder with its axis in the z direction.

To kick the ball, a torque was applied at the knee joint to cause the whole ankle to swing forward. The collision detection functions caused the foot to collide with the ball, transmitting a force to the ball and causing the ball to roll. Initially all bodies in the simulation were defined as cylinders, but due to a bug in PyODE, cylinders did not work properly with collision detection. If all bodies were defined as cylinders, the foot collided with the ball but no force was transferred during the collision and the ball did not roll. All cylinders in the model were changed to capped cylinders in order for the simulation to run properly.

Using the same control functions as the original simulations, a desired foot position was set before kicking the ball. Then data was gathered from the model to track the location of the ball and analyze how the ball trajectory varied with joint stiffness. There was very little friction between the ball and the surface it rolled on, so the ball rolled for a very long time. In the interest of time, the ball position after 2 seconds was recorded and is plotted in Figure 10.

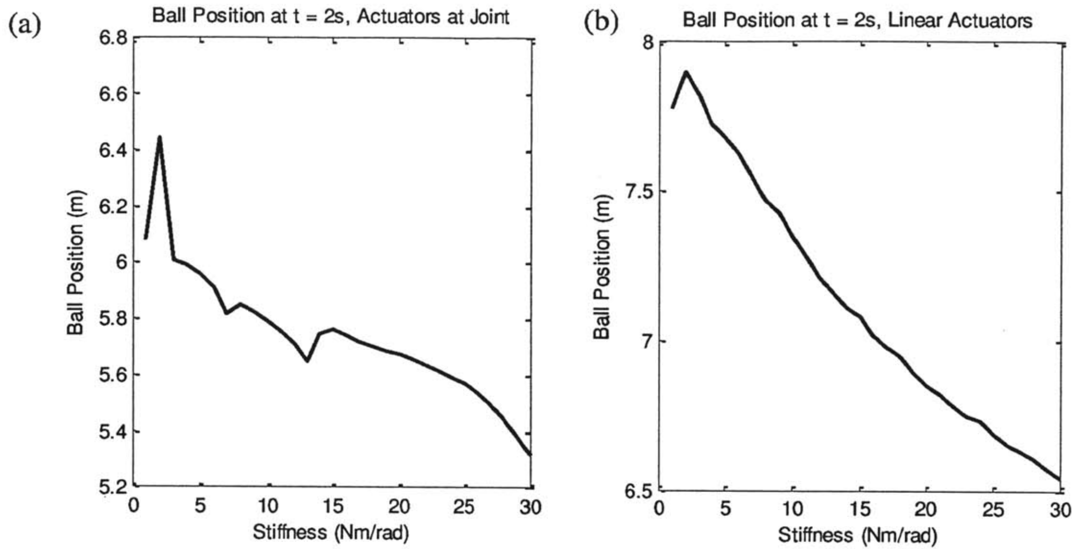


Figure 10: Final ball position as a function of stiffness for (a) actuator at the joint and (b) linear actuators. Ball started at x = 0.3048 m.

The curve in Figure 10a seems to have discontinuities. This may be due to the fact that as the foot swung, the angle of the ankle changed and so the controller added torque to the ankle to try to return to the reference angle. It is possible that the discontinuities in the graph are due to different impact angles or ankle torques during the simulations. The general trend for both models is that the ball will travel farther with less stiffness after reaching a peak distance with very low stiffness (about 3 Nm/rad). With lower stiffness the foot was in contact with the ball for a longer period of time than with higher stiffness. This meant that more energy was transferred during the collision between foot and ball and thus the ball traveled farther. However if the joint stiffness was too low, the collision moved the foot backwards instead of moving the ball forwards.

Changing the angle of the foot at impact or the point of contact of the foot with the ball changed the response of the ball to the kick. This simulation was run again changing the starting

location of the ball by moving it an additional 0.0672 m further away. This changed the contact point between the toe and the ball as shown in Figure 11.

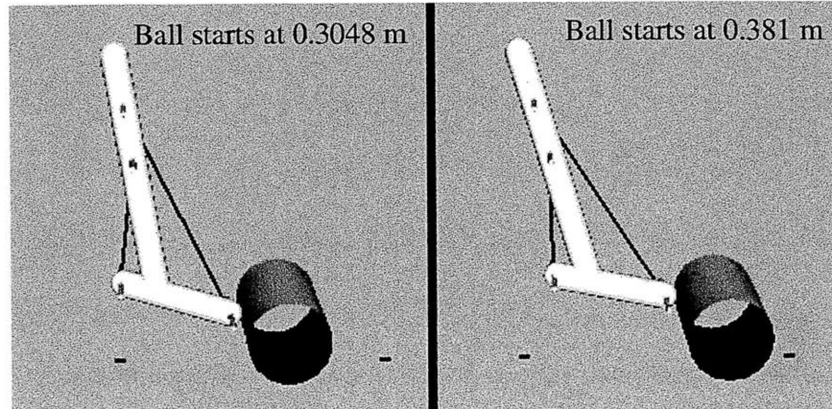


Figure 11: Contact point between foot and ball changes when ball starts in a different location. Ball at 0.3048 m (left) vs. ball at 0.381m (right)

The ball position at 2 seconds after kicking for the new starting configuration is shown in Figure 12.

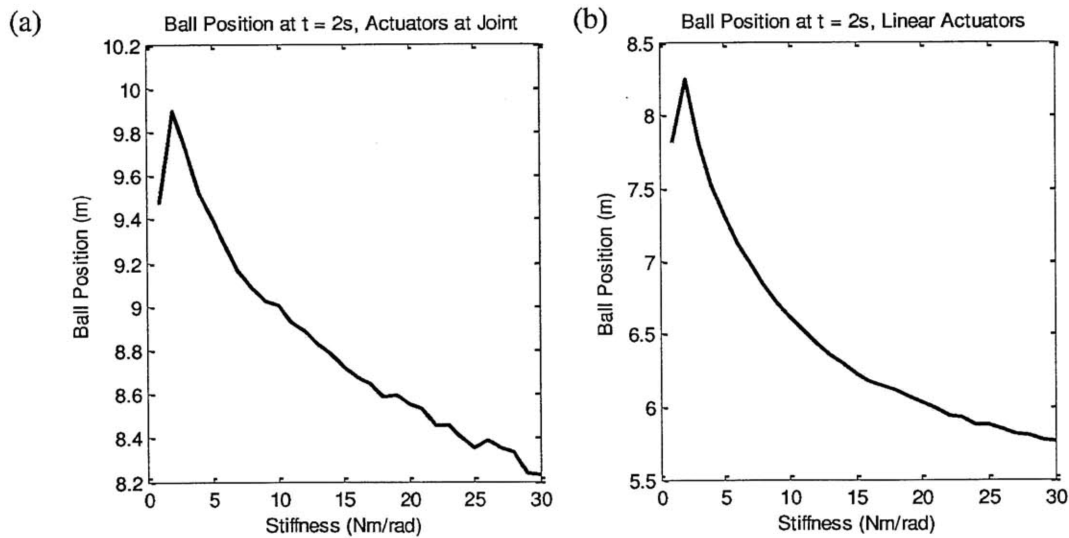


Figure 12: Final ball position as a function of stiffness for (a) actuator at the joint and (b) linear actuators. Ball started at x = 0.381 m.

The curve in Figure 12a is much smoother than that in Figure 10a. The ball also traveled farther in the second simulation where it started farther away. This suggests that ball position and foot impact point are significant factors in determining the motion of the ball after a kick. The ankle had more time during the kick to return to the reference angle than in the previous simulations with the ball closer. Figure 12b is very similar to Figure 10b but again it is a smoother curve. These simulations still show an optimal stiffness for kicking of around 3 Nm/rad so this may be constant despite changes in starting geometry. It would be interesting to measure the stiffness of a skilled soccer player's ankle during a kick to see if this is actually how a human would control their ankle during a kick.

4.3 Simulating Stepping

To simulate taking a step, the ankle was dropped onto the ground with the foot plantar flexed and dorsi-flexed. The ankle was dropped by removing the fixed joint with the environment at a specified time during the simulation. This meant that the shin was no longer constrained to remain vertical throughout the simulation and could rotate. The progression of the drop test is shown in Figure 13.

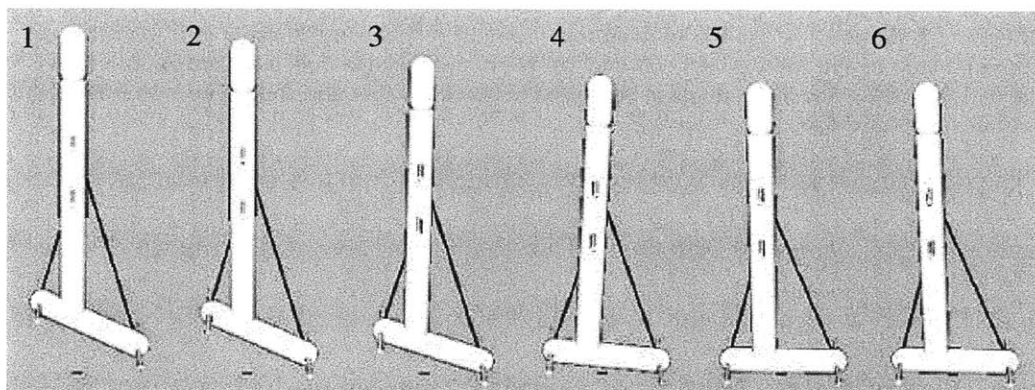


Figure 13: Progression of a drop test. In frame 1 the shin is stationary. Then the fixed joint is removed and the whole ankle falls. The foot makes contact with the ground (dotted line) in frame 3.

The joint angle between the foot and shin and the torque on the ankle joint were examined. In Figure 14 the response of the models to such a drop is shown.

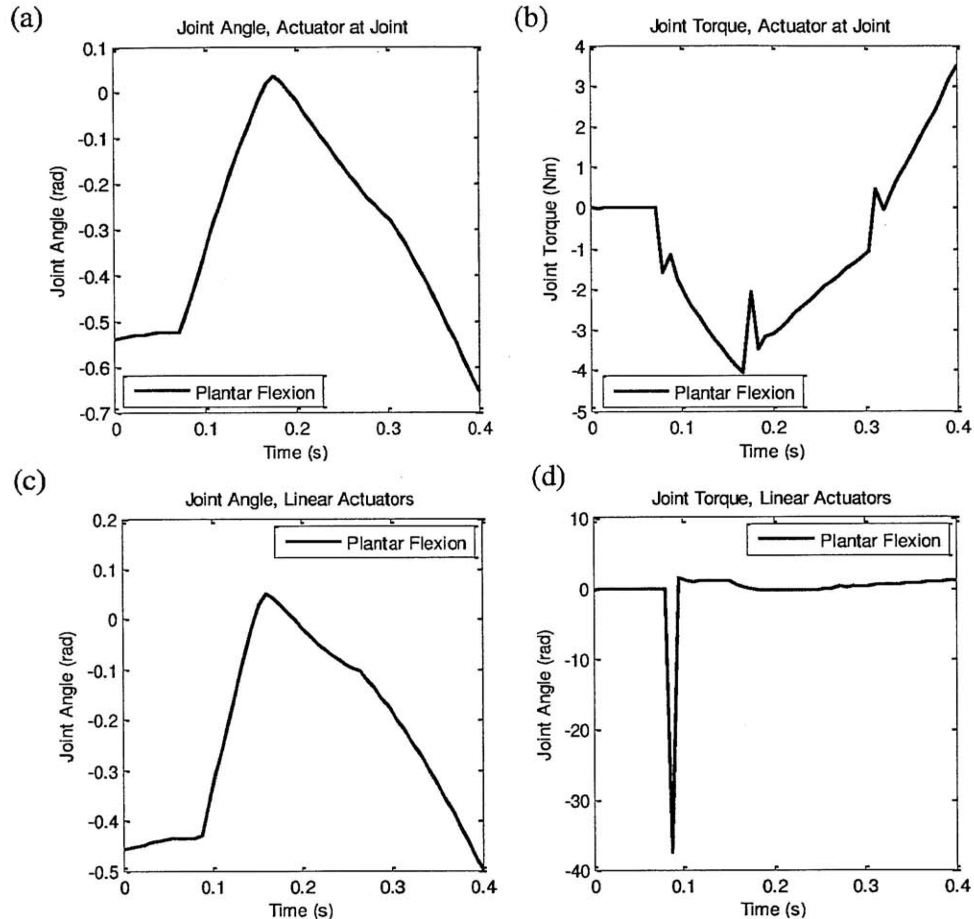


Figure 14: Response of models to drop test. (a) and (b) show ankle angle and joint torque for actuator at the joint model, (c) and (d) show ankle angle and joint torque for linear actuator model. The ankle angle is measured between the foot and shin. The shin is free to rotate during the drop.

The joint angle plots for the two models during this drop test are similar, as expected, but the joint torque plots are very different. This torque was recorded using the 'getFeedback' method in PyODE to evaluate the torque in the z-direction at the ankle joint during the simulation. Both plots show negative torque upon impact, but the linear actuator model had a sharp spike of great magnitude and the coaxial actuator model had a smaller amount of torque over a longer period of time. This may be a result of how the different actuators apply torque on

the ankle joint as nothing else in the two simulations is different. However it may also be due to a bug in PyODE's 'getFeedback' method.

It is possible that during walking or running the stiffness of the ankle is controlled so as to take advantage of weight acceptance during foot contact to move the knee to a posture appropriate for weight support. Drop tests were run on the ankle models with varying joint and muscle stiffness and the trajectory of the knee was analyzed. Figure 15 shows the trajectories of the center point of the knee for each model with varying joint stiffness.

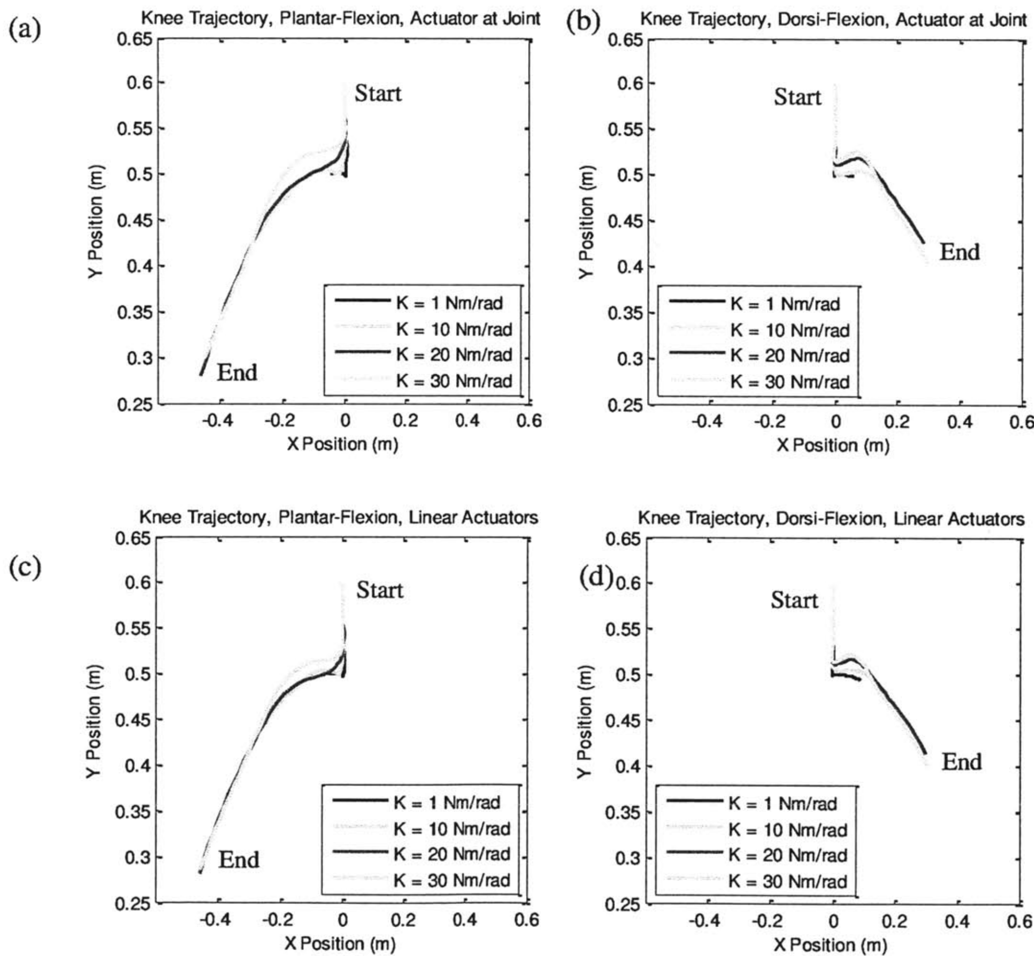


Figure 15: Response of models to drop test. (a) & (b) knee trajectory for actuator at the joint model, (c) and (d) knee trajectory for linear actuator model. Starting foot position and joint/muscle stiffness are varied.

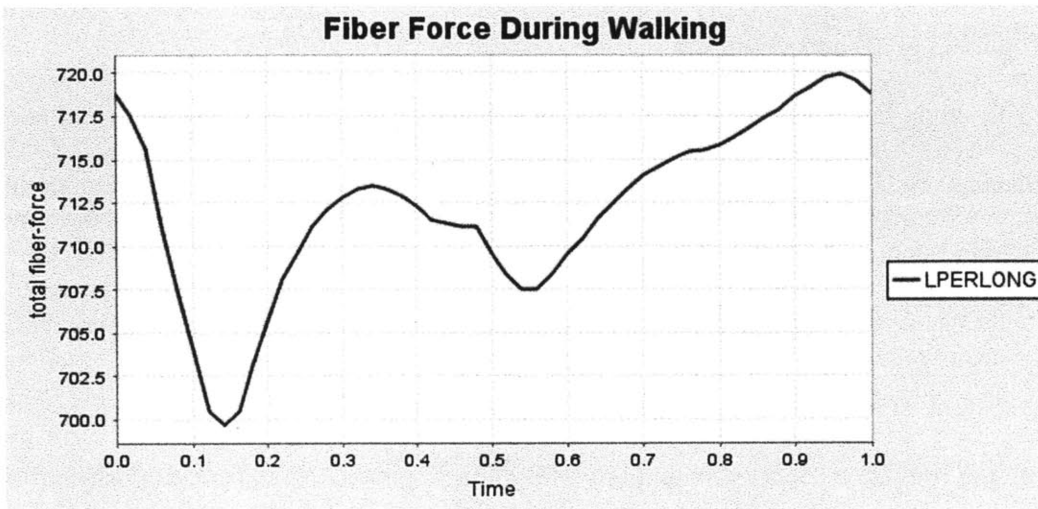
The simulations with the toe plantar flexed caused the knee to move slightly forward on impact and then fall backward; the simulations with the toe dorsi-flexed caused the knee to move slightly backward on impact and then fall forwards. The responses for both models to the drop tests were similar, which was expected. Since a normal walking motion involves dorsiflexion during stepping this motion makes sense. The stiffness used during walking might be that which minimizes the slight backward motion of the knee on impact. Sprinting involves plantar-flexion during stepping. The duration of ground contact in sprinting is also much shorter than in walking and it is possible that the next step is taken in a sprint before the backwards motion of the knee would occur. The ankle stiffness during sprinting might be that which yields the most forward motion of the knee to propel the runner.

4.4 Model Analysis in OpenSim

The OpenSim user interface (UI) contains a muscle editor that allows users to change parameters of each muscle in the model. A user may change any parameters of the muscles that are originally defined in the code for the model. However, running the motion with new parameters did not change the motion in any way, implying that the motion file imposed a specified time course for the motion of the model. Changing muscle parameters changed the muscle forces evoked by the motion.

A plotting tool exists in OpenSim to facilitate graphing of relevant model parameters. Figure 15 shows plots of the total fiber force in a muscle during the walking motion. Between the two plots a muscle parameter was changed. In Figure 16a the maximum isometric force in the left peroneus longus muscle was set to 756 N and in Figure 16b the maximum isometric force was set to 500 N.

(a)



(b)

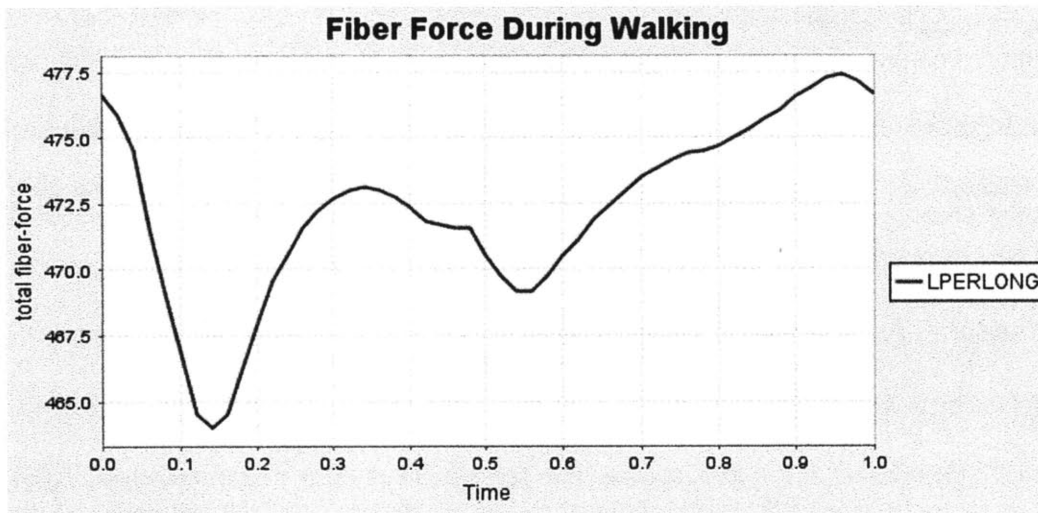


Figure 16: Plot of muscle force during walking motion for peroneus longus muscle. In (a) the maximum isometric force was set to 756 N and in (b) the maximum isometric force was set to 500 N.

5. DISCUSSION

5.1 Comparing ODE and OpenSim

This study has shown that both PyODE and OpenSim can be used to run simulations of biomechanical systems that would be difficult to create in conventionally popular simulation environments such as Matlab. While both programs can be used for essentially the same function, they have different features that likely make them more useful in different contexts.

Creating simple models in PyODE is quick and fairly easy if one is familiar with programming in Python. In this context 'simple' refers to a model with a small number of bodies and degrees of freedom and a low level of anatomical detail. A more complex model necessarily means more complex code and as a result errors are more likely to occur. OpenSim, while it follows the same principle that more bodies and degrees of freedom means a more complex model, appears to handle the increased complexity better than PyODE. The fact that it specifically includes code to define muscles makes OpenSim inherently less complicated than PyODE when it comes to increasing anatomical fidelity from actuators coaxial with a joint to linear actuators to simulate muscles. If a simulation requires a high level of anatomical detail in the model OpenSim is the better choice, but for simple models and simulations PyODE is adequate.

The user interface in OpenSim simplifies model and simulation analysis because it removes the need for detailed programming. If this were to be used in a setting like a problem set, a model and motion could be created in advance and made available to students for analysis. They need not have knowledge of C++ in order to work with the model. The benefit of PyODE is the ability of the user to quickly and easily change the parameters of a model or simulation.

To produce a new motion in OpenSim new simulation code must be written, as opposed to changing a few variables or lines of code in a PyODE simulation.

5.2 Potential Significance

Models and simulations such as the ones created using PyODE in this study can be used to analyze how environmental or neuromechanical factors affect the motion of biomechanical systems. Simulations of this type in conventional programs used to model rigid body dynamics are difficult or complicated to perform. If accurate models can be created for such simulations more insight may be gained into the way biomechanical systems behave. The simulated step and kick are very basic examples of motions that can be analyzed in this manner. These simulations can be made increasingly complex by adding degrees of freedom or additional limbs. The effect of parameters other than stiffness can be examined as well. The kick simulation showed that the results can change dramatically by simply changing the relative position of the bodies by a few inches. Manipulating these parameters in PyODE is trivial, so that many different simulations can be performed using essentially the same model with very little difficulty.

The benefits of OpenSim are that the user interface opens up the simulation environment to a broader group of people. Those who lack programming knowledge can still use OpenSim to run simulations and analyze models. There is great potential for using this as a teaching tool to illustrate the motion of biomechanical systems with predefined models that students can analyze in the framework of the user interface.

5.3 Future Investigation

In the future it would be interesting to push the limits of PyODE and generate more complex models and simulations than those created here. Another avenue of investigation would be to gather data from human subjects performing the same as simulated. This could then be

used to determine the accuracy of the simulation compared with reality. The models could be modified accordingly to more closely replicate the motion of a human ankle. It is also worth looking into OpenSim in more depth to determine exactly how much control a user has over the parameters of the muscles, both through the user interface and when programming a model. If the same level of control over the model exists in OpenSim as in PyODE then the existing user interface is a great advantage of OpenSim. However, it would be interesting to create a similar interface for simulations in ODE as well.

REFERENCES

[1] Smith, Russel. Open Dynamics Engine. 2007. http://ode.org/

[2] Neuromuscular Biomechanics. http://public.simtk.org/~ddelp/templates /final_03/ neuromuscular-biomechanic.html

[3] S. Delp, et al. OpenSim Developer's Guide. Stanford University, 2011.

[4] A. Roy, et al. Measurement of Human Ankle Stiffness Using the Anklebot, Proceedings of the

2007 IEEE 10th International Conference on Rehabilitation Robotics, June 12-15, Noordwijk,

The Netherlands.

[5] E. J. Rouse, et al. Design and Validation of a Platform Robot for Determination of Ankle Impedance During Ambulation, 33rd Annual International Conference of the IEEE EMBS

Boston, Massachusetts USA, August 30 - September 3, 2011

Appendix A: PyODE Code for Simulations

Update Muscle Force/Torque

Coaxial Actuator Model:

```
def update(self):
""" the switchboard for event-based control system org """
self.discretizeControl()

for j in self.joints:
      if j.name == "jAnkle":
            theta = j.getAngle()
            thdot = j.getAngleRate()
            if theta > 0:
                  K = 10
            else:
                  K = 10
            B = .3
            Tau_cmd = self.PD(th_ref, 0, K, B, 'jAnkle')
            j.addTorque(Tau_cmd)
```

Linear Actuator Model:

```
def update(self):
      """ the switchboard for event-based control system org """
      self.discretizeControl()

      a = 5*.0254
      b = 2*.0254
      c = 18*.0254 - .1
      d = 18*.0254 - .2
      l20 = sqrt(a**2 + c**2)
      l10 = sqrt(b**2 + d**2)
      dl2 = sqrt(a**2 + c**2-2*a*c*cos(pi/2 - thref))-l20
      dl1 = sqrt(b**2 + d**2-2*b*d*cos(pi/2 + thref))-l10

      for j in self.joints:
            if j.name == "rMuscle":
            l = -j.getPosition() + dl2
            v = j.getPositionRate()
            F = k*l - B*v
            j.addForce(F)

            if j.name == "lMuscle":
            l = -j.getPosition() + dl1
            v = j.getPositionRate()
            F = k*l - B*v
            j.addForce(F)
```

Changing Angle Simulation

(Code is within function that updates the world)

Coaxial Actuator Model:

```
th_ref = 0
counter += 1

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])

if counter >= 101:
      th_ref = pi/6

if counter >= 201:
      th_ref = -pi/6
```

Linear Actuator Model:

```
thref = 0
counter += 1

B = 12
ma1=(b*d*cos(angle)/(sqrt(b**2+d**2)+ragdoll.jMuscleL.getPosition()))
ma2=-(a*c*cos(angle)/(sqrt(a**2+c**2)+ragdoll.jMuscleR.getPosition()))
K = 12
k = K/(ma1**2 + ma2**2)

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])

if counter >= 101:
      thref = pi/6

if counter >= 201:
      thref = -pi/6
```

Adding Force Simulation

(Code is within function that updates the world)

Coaxial Actuator Model:

```
th_ref = 0
counter += 1

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])

if counter == 51:
    ragdoll.bFoot.addForce((0,300000,0))
```

Linear Actuator Model:

```
thref = 0
counter += 1

B = 12
ma1=(b*d*cos(angle)/(sqrt(b**2+d**2)+ragdoll.jMuscleL.getPosition()))
ma2=-(a*c*cos(angle)/(sqrt(a**2+c**2)+ragdoll.jMuscleR.getPosition()))
K = 10
k = K/(ma1**2 + ma2**2)

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])

if counter == 51:
    ragdoll.bFoot.addForce((0,300000,0))
```

Kick Simulation

(Code is within function that updates the world)

Coaxial Actuator Model:

```
th_ref = -pi/6
counter += 1

if counter >= 51:
    ragdoll.jKnee.addTorque(20000)

    angle = ragdoll.jAnkle.getAngle()
    jointangle.append(angle)
    feedback = ragdoll.jAnkle.getFeedback()
    foot_torques_Z.append(feedback[1][2])
    ball_pos = ball.getPosition()
    ball_pos_X.append(ball_pos[0])
```

Linear Actuator Model:

```
thref = -pi/6
counter += 1

B = 12
ma1=(b*d*cos(angle)/(sqrt(b**2+d**2)+ragdoll.jMuscleL.getPosition()))
ma2=-(a*c*cos(angle)/(sqrt(a**2+c**2)+ragdoll.jMuscleR.getPosition()))
K = 10
k = K/(ma1**2 + ma2**2)

if counter >= 51:
    ragdoll.jKnee.addTorque(20000)

    angle = ragdoll.jAnkle.getAngle()
    jointangle.append(angle)
    feedback = ragdoll.jAnkle.getFeedback()
    foot_torques_Z.append(feedback[1][2])
    ball_pos = ball.getPosition()
    ball_pos_X.append(ball_pos[0])
```

Drop Simulation

(Code is within function that updates the world)

Coaxial Actuator Model:

```
th_ref = -pi/6
counter += 1

if counter == 51:
       stop.empty()

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])
knee_pos = ragdoll.bKnee.getPosition()
knee_pos_X.append(knee_pos[0])
knee_pos_Y.append(knee_pos[1])
```

Linear Actuator Model:

```
thref = -pi/6
counter += 1

B = 12
ma1=(b*d*cos(angle)/(sqrt(b**2+d**2)+ragdoll.jMuscleL.getPosition()))
ma2=-(a*c*cos(angle)/(sqrt(a**2+c**2)+ragdoll.jMuscleR.getPosition()))
K = 10
k = K/(ma1**2 + ma2**2)

if counter == 51:
       stop.empty()

angle = ragdoll.jAnkle.getAngle()
jointangle.append(angle)
feedback = ragdoll.jAnkle.getFeedback()
foot_torques_Z.append(feedback[1][2])
knee_pos = ragdoll.bKnee.getPosition()
knee_pos_X.append(knee_pos[0])
knee_pos_Y.append(knee_pos[1])
```

Appendix B: Links to installation files

PyODE: http://pyode.sourceforge.net/

OpenSim: https://simtk.org/project/xml/downloads.xml?group_id=91