Marathon Walking Robot

Michael Tosto

212 ½ Delaware Ave.
Ithaca, NY 14850
(607) 351-9169
mat46@cornell.edu

May 2006

M&AE 429, 6 Credits (Fall '05 & Spring '06)

Faculty Advisor: Andy Ruina, T&AM and M&AE
Abstract

Improvements in mechanical, electrical and software systems were applied to a two dimensional bipedal passive dynamics based walking robot with the aim of achieving reliable and efficient long distance walking capabilities. Areas of improvement included the mechanical drive train, motor control and joint movement algorithms, startup routines, and stride control methods. Although the robot is not currently able to walk further than it could before these improvements, all have been steps which will aid any future work to improve distance capabilities and system robustness.

Introduction

The philosophy behind passive-dynamics based robotics differs from the “conventional” approach because rather than using the brute force method of precise actuation on many degrees of freedom to mimic a human’s gait, a more natural perspective is used. A robot’s basic mechanical characteristics can be designed to create a system that requires only a very small amount of force input to maintain a stable walk cycle. Pendulum-like motion of the legs can be exploited, as humans do, to achieve walking motion without excessive wasted energy [1].

Construction of this robot began in 1998 [3] and progressed through various stages of actuation, design and control until Matt Haberland was able to achieve a stable walk cycle during the summer of 2005 [4]. This was done by implementing a state based control algorithm in which sensor input was used to determine which of three possible general situations described the robot’s configuration. Because of the relative similarity between the inner and outer legs in terms of the actions necessary for walking, no differentiation was made between the two in terms of state. To compensate for asymmetries between the two legs’ mass characteristics, different gains were used in controlling the hip swing of the outer legs as compared to the inner legs [4]. A picture of the robot is shown below in figure 1.
The robot’s configuration at any point in time can be described in terms of three degrees of freedom: two ankle joints and one hip joint. Each of these three joints uses a DC motor for actuation, and a potentiometer for sensing of angular position. The mechanism used to attach the ankle joint potentiometers is shown in figure 2. As either foot rotates, its chain will turn a sprocket which is attached to a potentiometer. In addition to the potentiometers, two limit switches on the feet signal whether or not each foot is in contact with the ground. These switches are visible in figures 1 and 3. In addition, the analog signal from each potentiometer is routed through a differentiator circuit to find the angular velocity of each joint. With the addition of a new interface board, sensors also provide the current through all three motors. To make the robot walk, information from all eleven of these sensors must be continually interpreted and used to find an appropriate PWM duty cycle each for the three motors.
While Matt’s work showed that this design can walk, it also exposed limitations of the robot and necessitated that other improvements be made before the goal could be realized. Hardware weaknesses included flimsy drive chains, messy wiring, slow motor speed controllers, and backlash in potentiometer readings among other problems. Software limitations included a difficult starting mechanism, weak foot and hip motion control, fully open loop stride control, and motor control based on PWM duty cycle with rather than torque. To address these concerns, improvements were made to the system this year which will lead to greater walking ability in the future and less time spent on fixing problems as they occur.

Beyond the immediate goal of building a robot capable of walking long distances, work in the field of passive dynamics based robots might be applied to prosthetics and physical therapy, as well as more general robotic locomotion.

**Methods and Results**

**Potentiometer Calibration Function**

The first issue we addressed was calibration of the potentiometers to eliminate the need for hard-coded changes in the control code each time the chain links slipped and the potentiometer was not in the same predefined initial position. To accomplish this, a new startup state was added to the robot’s existing state machine control. This initial state was placed before “State I” (a launch preparation state), and is used to get a baseline reading of potentiometer values at a known machine configuration. To do this, the robot now goes into “StateCal” as soon as the controller is turned on, where it will stay until satisfactory calibration has been performed. While in this state, the user manually moves the robot’s feet to the retracted position where they are pressed all the way up against their stops and are pointing vertically upward. After this, limit switches on each set of feet are pressed in series to signal that the computer should read potentiometer values at this known orientation. Recalibration of each ankle can also be performed by pressing either limit switch again after initial calibration, while still in the calibration state. The final reading is then saved, and all future target locations for actions such as ankle preparation before heel-strike, and ankle push-off are based on relative angular differences. To alter the actual motion of the
ankles and hips, the user need only change the magnitude of these relative differences and then perform calibration when powering up the robot.

Because potentiometers read an absolute angular position based on their range of rotation, they are not able to turn many revolutions the way an optical encoder can. This means that each potentiometer must be sufficiently far from its physical stop before operation, so that the motors do not cause damage to the hardware by forcing it beyond the intended operation range. To check for this, absolute locations of the potentiometers are output through the serial cable as each limit switch is pressed, and values may be checked via the terminal window when connected to a computer. If calibration is being performed while away from a computer, the potentiometer may be manually moved independently from the chain and placed in a safe orientation. For the code used to perform these actions, see appendix A.

After the initial calibration has been performed, pressing either of the heel limit switches again will recalibrate to the new current position and forget the old calibration. To indicate the end of calibration, a push button was added and placed on the main front mounting board next to the on/off switch and battery charging plugs. Pressing this button will signal the controller to continue on to the main operation states where walking actions take place. This switch was connected to digital interrupt 3 through the new interface board created by Justin Webb.

Another method considered for performing calibration was to automatically read ankle positions whenever the controller is turned on and assume that this represents the retracted configuration. However, this requires that the feet are always completely retracted before powering on the controller, which is not easy to do with a lone operator. Also, the automatic method does not allow for subsequent recalibrations, and therefore would exclude one of the major reasons for having calibration functionality. Because of these considerations, the original method described above was chosen for its greater flexibility and ease of use.

Drive Train Improvements

After multiple instances of breakage in the existing chains, their replacement became necessary. The original chain material was plastic with a metal core, chosen for its light weight. However, when a similar chain with greater tensile strength could not be found, ANSI #25 steel roller chain was selected as the replacement. In addition to its greater strength, this type of chain also has the advantage of being a standard item, and therefore has greater widespread availability. To tension the new chains, springs were inserted between links on the side of the chain that does not bear the weight of the robot. A list of components used in the replacement process is shown here in table 1, along with their McMaster-Carr part number and price. After replacing both of the foot chains, the resulting setup is shown below in figure 3.
Table 1 – Chain Replacement Parts List

<table>
<thead>
<tr>
<th>Description</th>
<th>Part Number</th>
<th>Items/Unit</th>
<th>Price/Unit</th>
<th># of Units</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprockets - ANSI #25, 10 tooth, 1/4&quot; bore, 1/2&quot; hub</td>
<td>2737T2</td>
<td>1</td>
<td>$3.48</td>
<td>9</td>
<td>$31.32</td>
</tr>
<tr>
<td>Chain connecting link - ANSI #25</td>
<td>6261K108</td>
<td>1</td>
<td>$0.70</td>
<td>12</td>
<td>$8.40</td>
</tr>
<tr>
<td>Chain add-connect link - ANIS #25</td>
<td>6261K105</td>
<td>1</td>
<td>$1.46</td>
<td>4</td>
<td>$5.84</td>
</tr>
<tr>
<td>Springs - 1&quot; extension @ 7.21 lbs, 2&quot; long, 3/8&quot; diam.</td>
<td>94135K14</td>
<td>3</td>
<td>$4.68</td>
<td>1</td>
<td>$4.68</td>
</tr>
<tr>
<td>Springs - 1&quot; extension @ 4.33 lbs, 1.5&quot; long, 5/16&quot; diam.</td>
<td>9654K126</td>
<td>12</td>
<td>$9.53</td>
<td>1</td>
<td>$9.53</td>
</tr>
<tr>
<td>Stainless steel rod, 1/4&quot;D, type 304, 6' length</td>
<td>8934K13</td>
<td>1</td>
<td>$17.24</td>
<td>1</td>
<td>$17.24</td>
</tr>
<tr>
<td><strong>Total Cost</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$77.01</td>
</tr>
</tbody>
</table>

Figure 3 – Finished Foot Chain Replacement

To check the amount of mass added to the robot from chain replacement, weight of the original components was compared to the weight of the new chain components. This was an important step because one of the primary reasons for not using a metal roller chain from the beginning was weight considerations. Upon comparison, about 600g of extra mass would be added to the robot from the heavier steel chains, as shown in table 2.

Table 2 – Chain Mass Comparison

<table>
<thead>
<tr>
<th>Unit Masses</th>
<th>Current</th>
<th>#25 Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprocket</td>
<td>6 g each</td>
<td>11.2 g each</td>
</tr>
<tr>
<td>Chain</td>
<td>0.139 g/cm</td>
<td>1.69 g/cm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Amount Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprocket</td>
</tr>
<tr>
<td>Chain</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total Masses (g)</th>
<th>Current</th>
<th>#25 Chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sprocket</td>
<td>54</td>
<td>67.2</td>
</tr>
<tr>
<td>Chain</td>
<td>53.5</td>
<td>650.7</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>107.5</td>
<td>717.9</td>
</tr>
</tbody>
</table>

| Difference       | 610.4   |

In addition to these changes, the hip drive chain was also modified for better performance. First, the original idler bolt used to tension the chain was removed to reduce unnecessary friction. To
tension the chain, 1 link was removed and both the motor mount and the potentiometer mount were machined to have vertical slots instead of circular screw holes. This allowed vertical motion of the motor until the chain was taught, where it was then clamped in place. However, this solution proved to be unsatisfactory so Justin Webb designed and machined a new combined motor/potentiometer mount. With the slotted mounts, torque from the motor would loosen the mount over time, adding undesirable slack to the chain which in turn reduces the accuracy of hip motion. To solve this problem, the new mount has a lip extending over the top of the frame to prevent slippage. Vertical position is still adjustable using shims placed under this lip, but after being set, it cannot slip as the old one could.

Foot Control and Simulation

To improve the original method of foot control, a PD control scheme was implemented to take the place of simple proportional control. Adding derivative terms to the control effectively adds damping to the system and will therefore reduce oscillation about the desired position if given a step input. The general form for PD control as used in the robot is shown here in equation 1 [2].

\[
\tau = K_p (\theta_d - \theta) - K_d \dot{\theta}
\]

*Equation 1 - General PD Control*

This type of control was made possible by differentiating the signal from each potentiometer, which results in a signal representing joint angular velocity. Because of the short amount of time available for ankle retraction and preparation, it is important that the control has as little overshoot and oscillation as possible, ensuring that the foot always hits the ground at the same angle. Although equation 1 describes torque as the control parameter, PWM duty cycle values were used instead because there was no available way to directly specify motor torque. This usually produces the expected responses, but because duty cycle is not proportional to torque, it is not strictly correct. To address this problem, torque control methods were later written and applied.

To choose specific control gains, arbitrary values were inserted and iteratively modified in response to observed joint motion. A LabView data acquisition program written by Ko Ibara greatly aided this process by plotting the ankle positions over time from parsed output of “printf” statements. By inserting a print statement at the beginning of the main program loop, values are sent over a serial cable at each iteration and received by LabView. Results of plotting the inner and outer foot motion before and after tuning the PD gains are shown in figures 4 through 6.
Figure 4 – Inner Foot Motion, After

Figure 5 – Outer Foot Motion, Before
For evaluation of these results, a rough simulation of the foot-motor system was constructed using Simulink. To perform the simulation, the first step was to identify a general differential equation of motion for a permanent magnet DC motor. Equation 2 below shows the relationship between torque, current, voltage and speed [6].

\[ V - K_m \theta = R_a i, \quad i = \frac{\tau_m}{K_a} \]

*Equation 2 – DC Motor Equation of Motion*

In this equation, V represents voltage, \( K_m \) is the DC motor constant, \( R_a \) is the armature resistance, \( \tau_m \) is the motor torque and \( K_a \) is the armature constant. To find torque applied by the motor, it is assumed to be produced by the inertia of the foot for this estimation. The foot diagram and accompanying equations below show a conversion of coordinates and substitution into equation 2 to eliminate the current variable.
Robot Foot:

\[ F_l = \tau_m \quad \quad F = ma = m \ddot{\theta} \]

\[ \tau_m = ml^2 \ddot{\theta} \]

\[ V - K_m \theta = R_a \frac{ml^2 \ddot{\theta}}{K_a} \]

\[ V(t) = \alpha_1 \theta(t) + \alpha_2 \theta(t) \]

\[ \alpha_1 = \frac{R_a ml^2}{K_a} \]

\[ \alpha_2 = K_m \]

After obtaining the final relationship between voltage and angle shown in equation 3, the Laplace transform was taken to convert the equation into the frequency domain, and rearranged to get a transfer function.

\[ V(s) = s^2 \alpha_1 \theta(s) + s \alpha_2 \theta(s) \]

\[ \frac{\theta(s)}{V(s)} = \frac{1}{s \alpha_1 + \alpha_2} \]

Equation 4 – Motor Model Transfer Function

This resulting transfer function was then inserted into a Simulink model along with a PID controller (with integral gain set to zero) to simulate the motor-foot system. Figure 8 below shows this model with and without a transport lag introduced into the feedback signal. Because numeric values of the two constants \( \alpha_1 \) and \( \alpha_2 \) were not known, they were both assumed to be 1 to give a general sense of how a similar system might behave. Also, because the precise amount of time lag due to backlash was not known, a value of 0.1sec was chosen to represent a generalized delay. Along with this, a step input of 1 was selected to represent a general change in desired configuration. Gains were set at \( K_p = 20 \) and \( K_d = 12 \) for both simulations so that results could be compared to each other.
Running these two simulations yielded the results shown in figures 9 and 10 below. With transport delay included the results closely mimic the actual response observed in figure 6, suggesting that backlash induced time delay contributed to the oscillatory motion observed. However, backlash was significantly reduced after Justin Webb replaced the potentiometer mounting mechanism to eliminate unnecessary gears. This upgrade in conjunction with Ko Ihara’s custom h-bridge boards served to greatly reduce settling time and overshoot of the foot motion, as shown in figure 11.
Figure 9 – Simulation With Transport Delay

Figure 10 – Simulation Without Transport Delay

Figure 11 – Latest Outer Foot Motion
Hip Swing Algorithm

One of the problems that has most directly limited the robot's ability to walk has been its inconsistent hip swing. Based on simulation work by Gregg Stiesberq, if the hip could swing through to a set angle and solidly lock until contact with the ground, it would be much easier to achieve a stable walking motion. With this in mind, the hip swing algorithm was rewritten in an attempt to make the stride more predictable. Rather than applying linear PD control throughout the entire swing with gains changing at the midpoint, a more non-linear approach was used. At the beginning of its swing, the hip is given a slight push with the motor to start it moving. After reaching a set angular displacement, and for the majority of the motion, it swings freely with no torque applied by the motor. As it gets close to the target configuration, reverse torque is applied over a set angular range, after which it switches to high gain linear PD control. By this time the hip will be very close to its desired angle and have small angular velocity, therefore minimizing the amount of oscillation and overshoot occurring due to the PD control.

A problem with this algorithm was that the “swing state” divisions described above (defining when to perform different actions) were based solely on position of the hip. This required the assumption that the swing leg is moving forward at all times until reaching the target. If the reverse torque is too much, or if its range of application is too large, the leg will not make it past this point, and will fall back to the neutral position. To eliminate this problem, the swing states were divided into a case structure similar to the main state machine control where progression can only occur forward from one state to the next. Using this method, the exit conditions of the reverse torque state were changed to be zero angular velocity of the hip. In effect, this guarantees that before switching to high gain linear control, the hip is always momentarily at rest, and close to the set position. For the code used to perform this swing, see appendix B. Figure 12 below shows the current hip swing while the outer legs are held stationary and the inner legs are released from rest in back.
Before using this method, another hip swing algorithm was attempted, but abandoned because of inefficiency and clumsy control. After noting that the motor is definitely strong enough to support the weight of either pair of legs at a given position (as walking simulations recommend), the original hip swing method was modified so that the hip motor switches to a constant PWM duty cycle after reaching the desired position, to effectively lock the hip against its physical stop. However, because the motor was often stalled against the physical stop, it would heat up very rapidly when used in this way. Also, the reliance of this method on physical stops was a disadvantage because unnecessary collisions waste energy, and are difficult to consistently control.

Another method of hip control that could be used in the future would explicitly take into account the mass properties of the legs to create a swing that more closely matches the natural pendulum motion of the legs. If this could be accomplished, the efficiency of the hip swing would be improved, and could still maintain consistency.
Stride Mapping

To get a better understanding of the robot’s stride, as well as areas that need improvement the LabView data acquisition software was used to track each of the three joint angles over time. The resulting plot of the walk cycle shown below in figure 13.
By examining this type of plot, it is much easier to identify individual problems with the walk cycle and take action to correct them. For example, by looking at the hip angle in figure 12, it appears that one reason the robot may have fallen in this case is the inner leg swing becoming progressively smaller each step. After it became small enough, the robot could no longer continue with the next step.

**Dynamic Stride Control**

Rather than running the same set of instructions at every step to move the joints based on predefined configurations, the next action can be found using characteristics of the last step. One instance of this is in the ankle push amount required. To determine when to retract the ankle and swing the leg forward, the robot uses a target angular position. However, if the last step was too small, the robot will still use this same target position and fall over frontward because it waits too long before swinging the rear legs forward. If the legs are spread more than expected, the ankle push amount should be greater than expected, but if the legs are spread less than expected the ankle push should be less. In the neutral position with both legs parallel, there can be zero push before causing the robot to fall forward. This condition was used along with the “normal” full push amount to find an appropriate push at each step. To do this, the actual hip separation is calculated and found as a percentage of the desired separation. Because 0% separation requires 0% push, it was assumed that 50% separation would require 50% push, and so on. For the modified ankle push routine, see appendix C.

Another problem with the ankle push action was occasional stalling of the foot, which caused the robot to come to a stop. To counteract this, a small amount of hip actuation was added during ankle push to bring the hips together, therefore helping the ankle motors complete their push action. If the ankles are ever unable to reach their target push position and reach zero velocity, the robot can also be set to return to an initial state where both of the ankles are locked and the hips are spread. From this configuration, the robot can then be restarted as if it were just beginning to walk.

**Start/Stop Functionality**

To eliminate the uncertain element of operator interaction in starting the robot, a new launch method was developed that is able to start the robot from rest and enter the walk cycle by simply flipping a switch. This was also extended to allow the robot to stop while walking by flipping the same switch back to the “stop” position. With this type of launch mechanism, it becomes easier to debug the robot’s walking because the extra variable of user startup is removed. Additionally, this type of startup does not require the same intuitive “feel” and can therefore be executed by any operator. Because the switch is simply connected to digital interrupt 4 which changes a global variable when triggered, this same global variable could be written using a wireless communication program to remotely control the robot’s motion.
To connect the switch used for signaling the stop and go commands, an IDC connector was used so that it can be removed when not in use, and easily replaced if necessary. Figure 14 below shows a diagram of the wiring used to connect this switch.

Figure 14 – Go/Stop Switch Wiring

The vertical bar represents the screw connection strip on the interface board, with a shared ground used between the go/stop switch and the calibration button.

After the controller is turned on and calibration is complete, the robot will enter State I where it performs the same AnklePrepare function as it did before. However, when one of the limit switches is pressed, the robot will go on to “StateLock” where both of the feet are locked and the hip is free to swing. From this state, the robot will proceed to “StateRock” whenever the switch is triggered, and perform the necessary actions to complete the first step. After this, the robot will enter State III and continue on with the normal walk cycle. A flowchart of this progression is shown below in figure 15.

- 16 -
To make the first step from rest, the robot will first push with whichever foot is in back. If this foot is able to reach the retraction position with sufficient positive angular velocity, it indicates that the robot has enough forward momentum to swing the rear legs forward and complete the first step. However, if the rear foot is not able to reach this position, it will stop pushing whenever zero velocity is reached, and return to the locked position. After this, the front foot will push until it either reaches zero velocity or reaches a cutoff location to ensure the robot does not fall over backward. Next, the front foot will stop pushing and return to its respective locked position. The robot is now back where it started and will push with the rear foot, but with greater initial forward momentum. This forward and back rocking motion will continue until enough momentum is built up and the robot can make its first step. Figure 16 shows a flowchart of execution within “StateRock”, and figure 17 shows a plot of joint angles over time during the rocking motion. For the complete code used for rocking startup, see appendix D.
Figure 16 – StateRock Execution Flow
In figure 17, the outer leg starts out in back, and is the first to push. Initially the “bump” representing ankle push is relatively small, but grows in height (amount) and width (duration) over time, until eventually it reaches the required location to take a first step. Examining the hip angle, there is a slight bump just after the second push with the front foot. This is because as the robot falls forward, the hips have a tendency to fall together. To counteract this, a small amount of hip actuation was added during the entire routine to keep the hips spread the correct amount. In this case, the hips begins to fall together, but are quickly restored to their previous location.

Finding the cutoff location for pushing with the front foot is done in a similar manner as the dynamic ankle push. By comparing the current hip angle to the desired hip angle, the full push value is scaled accordingly. However, rather than simply performing linear scaling between zero and the full value, interpolation is done based on an arbitrary number of “set points” at hip spread percentages. The program decides which in which region the robot currently falls, and interpolates between the two surrounding predefined push amounts. The code used to do this can be found in appendix E.
Torque Control of Motors

To enhance control of the robot’s joints, a shift was made to use torque as the motor control parameter rather than PWM duty cycle. With the addition of the new electronics interface board and h-bridges, sensors provide the ability to use the current through each motor as a feedback signal. The benefit of this is that control can then be based upon motor torque which is proportional to current. Increasing the duty cycle has the effect of increasing the overall voltage “visible” to the motor, which in turn increases the motor’s speed. Although using duty cycle as the control parameter can work, it is more effective to perform PD position control based on motor torque, as noted in equation 1.

To achieve the goal of controlling based on torque, a desired torque is found from equation 1 using the desired position, actual position, and actual velocity. After this, a second PD control loop uses actual current, desired current, and current change (an approximation of $\dot{I}$) to find an appropriate duty cycle to achieve the target current at any given time.

$$PWM = K_p (I_d - I) - K_d (I - I_{prev})$$

Equation 5 – PD Control for Current/PWM

This effectively makes all of the user programmed controls one level higher than they previously were – to move the foot to a target position the function “PD2Torque” is called to find the desired torque. Using the current sensing capability, the controller then uses the “Torque2PWM” function to find the correct PWM value to achieve the target torque. The net effect of this is two nested control loops using feedback to ultimately find a duty cycle that should be sent to the motors at each point in time. For the complete code used to do this, see appendix F.

Other benefits of current sensing include the ability to ensure that the motors never exceed their maximum allowed current, as well as ensuring that motor torque never becomes too high in a given situation. For example, relatively little torque should be required during ankle retraction, and therefore a current limit could be enacted for this specific action to make sure the feet are never pressed against the physical stops in an attempt to reach an impossible position. Also, using this method will decrease the robot’s dependence on battery voltage for functionality. Rather than setting a given duty cycle and using it regardless of battery voltage, the duty cycle will be increased as necessary to reach the correct torque. This means that performance of motor control should not decrease over time and become “sluggish”, up until the battery simply cannot provide the needed amount of power.

Other Concurrent Work

In addition to the work described in this document, other members of the team working concurrently made the following important improvements to the robot. To replace the original electronics interface board, Justin Webb designed and created a new board that allows for easy use of 10 pin connectors and integrates the new h-bridge boards created by Ko Ihara. Justin also installed floating potentiometer mounts to reduce backlash in sensor readings, as well as creating
a new mounting bracket for the hip motor and potentiometer. In addition to designing the h-bridge boards, Ko also designed a custom brain board that holds a microcontroller, switching power regulator, 3 axis MEMS accelerometer, and 3 single axis MEMS gyros. While still in the initial stages of integration into the Marathon Walking Robot, this board’s integrated sensors, speed, and flexibility could be very useful. A chart detailing necessary connectors for the “KoBrain” is shown in appendix G. Andre Harrison has been working on developing a wireless communication system that would permit efficient two way communication between the microcontroller and an operator. This would allow for real-time plotting of joint variables, issuing of commands such as “go” or “stop”, as well as changing gains and target positions without reloading the code each time. Sarah Bates has designed and installed a ratchet mechanism for the outer feet that allows positive angular motion, but locks against retraction. This would greatly reduce the amount of power required from the motors because they would no longer be required to support the robot’s weight during each step. Complementing the ratchet, Matt Coryea has designed a new version of the feet that would improve on those designed by Zim Harry [5] and allow better ground clearance when swinging the hips. Additionally these feet would direct normal force from the ground through the shaft, and also provide more reliable sensing of contact with the ground.

Conclusion

After performing the improvements detailed here, the mechanical and software systems of the Marathon Walking Robot show greater reliability, durability and robustness. A potentiometer calibration function eliminates the need to hard-code changes in angular positions. PD control is now used for motion of the feet, while stronger steel roller chains deliver torque from the motors to the foot shafts. Using stride mapping, the hip swing can be identified as an important factor preventing the robot from walking farther. To improve the hip swing method, a nonlinear approach was used in an attempt to rigidly lock the hip at its target location after swinging through. Further stride reliability is achieved using an adaptive heel push method which uses the robot’s previous step as input to determine the amount of push-off required for the next step. The uncontrolled factor of operator interaction on startup is also eliminated by using an automatic start/stop routine which rocks the robot forward and backward until it is able to take its first step. Current sensors make torque based control possible, and also diminish the effect of reduced battery voltage on overall performance of the robot. The enhancements described here have improved the overall durability and flexibility of the Marathon Walking Robot and will aid future work toward the goal of efficient long distance bipedal walking.
References


Special thanks to Andy Ruina, Jason Cortell, Engineering Learning Initiatives, and all other graduate and undergraduate students involved for making this project possible.
Appendix A – Calibration Code

/****************************************************************************
* FUNCTION NAME: PotCal
* PURPOSE: Re-define base potentiometer values based on readings taken
* CALLED FROM: State Machine Function
* ARGUMENTS: InnerOffset, OuterOffset
* RETURNS: none
****************************************************************************/

void PotCal(int offset, char motor)
{
    if (motor == IN)
    {
        RetractedIn = RetractedIn + offset; // Inner Pot. reading, retracted position
        PreparedIn = RetractedIn + 330; // Inner Pot. reading, prepared position
        ToeOffIn = PreparedIn + 50; // Inner Pot. reading, Toe Off position
        InnerCalFlag = 1;
    }
    else if (motor == OUT)
    {
        RetractedOut = RetractedOut + offset; // Outer Pot. reading, retracted position
        PreparedOut = RetractedOut + 210; // Outer Pot. reading, prepared position
        ToeOffOut = PreparedOut + 40; // Outer Pot. reading, Prepared position
        OuterCalFlag = 1;
    }
}

// Initial Calibration State
/****************************************************************************
* Move the inner and outer ankles to their "retracted" positions, *
* push the limit switch on each corresponding ankle, and base pot. *
* positions will be reset to the currently read positions. *
* Press the *
****************************************************************************/

case StateCal:
{
    // Reset base inner potentiometer reading if inner switch pressed
    if (LimitHeelIn == CLOSED & LimitHeelOut == OPEN & InnerCalFlag == 0)
    {
        // Re-calibrate inner ankles
        PotCal(Potentiometer(IN), IN);

        // Display values
        printf("Inner Ankle Position: %d\n", RetractedIn);
        printf("Inner Ankle Speed: %d\n", (int)Differentiator(IN));
        printf("PreparedIn: %d\n", PreparedIn);
    }

    // Reset base outer potentiometer reading if outer switch pressed
    if (LimitHeelIn == OPEN & LimitHeelOut == CLOSED & OuterCalFlag == 0)
    {
        // Re-calibrate outer ankles
        PotCal(Potentiometer(OUT), OUT);

        // Display values
        printf("Outer Ankle Location: %d\n", RetractedOut);
        printf("Outer Ankle Speed: %d\n", (int)Differentiator(OUT));
        printf("PreparedOut: %d\n", PreparedOut);
    }

    // Proceed to State I if button is pressed
    // If calibration of inner and outer ankles has not been done, default values will be
    // used.
    if (CalSwitch == CLOSED)
    {
        printf("Done on to state I\n\n");
        State = StateI; // Assume State I
    }
    break;

    ...[remaining cases omitted]...
void HipSwing (void)
{
    // Initialize other variables needed
    int hipRelPos;
    int inSwinging;
    int hipPos = Potentiometer(HIP);
    // Initialize relative location variables
    int phase12;
    int phase23;
    int phase34;

    // Set a variable that indicates if the inner leg is swinging forward through the overall
    "HipSwing"
    if ((inback == IN & State == StateIII) | (inback == OUT & State == StateIV))
    {
        inSwinging = 1;
    }
    else if ((inback == OUT & State == StateIII) | (inback == IN & State == StateIV))
    {
        inSwinging = 0;
    }

    // Set _RELATIVE_ locations for phase divisions
    if (inSwinging==1)
    {
        phase12 = -50;
    }
    else
    {
        phase12 = -30;
    }
    if (inSwinging == 1)
    {
        phase23 = (SwingIn-HIPSWITCH)-10;
    }
    else if (inSwinging == 0)
    {
        phase23 = (HIPSWITCH-SwingIn)-10;
    }
    phase34 = phase23/2;

    // Find current relative hip position in relation to hipswitch
    if (inSwinging == 1)
    {
        hipRelPos = hipPos - HIPSWITCH;
    }
    else if (inSwinging == 0)
    {
        hipRelPos = HIPSWITCH - hipPos;
    }

    // use "case" structure to avoid backtracking
    switch(HipSwingPhase)
    {
    case 1:
    {
        // Send full "forward" power during Phase 1 (to push rear legs toward the
        front)
        if (inSwinging == 1)
        {
            SendPWM(HIP,FULL_FORWARD/2);
        }
        else if (inSwinging == 0)
        {
            // - 24 -
        }
SendPWM(HIP,FULL.Reverse);
}

/\ Exit condition
if (HipRelPos > phase12)
{
  HipSwingPhase = 2;
}
//printslow(HipSwingPhase);
break;

} case 2:
{
  // Send neutral during Phase 2 (let it coast for a while)
  SendPWM(HIP,0);
  // Exit condition
  if (HipRelPos > phase23)
  {
    HipSwingPhase = 3;
  }
  //printslow(HipSwingPhase);
  break;
}

} case 3:
{
  // Send full "reverse" power during Phase 3 (to slow down the legs as they
  // approach final pos.)
  if (inSwinging == 1)
  {
    SendPWM(HIP,FULL.Reverse/2);
  }
  else if (inSwinging == 0)
  {
    SendPWM(HIP,FULL.Forward/2);
  }
  // Exit condition
  if (Differentiator(HIP) > -3 & Differentiator(HIP) < 3)
  {
    HipSwingPhase = 4;
  }
  //printslow(HipSwingPhase);
  break;
}

} case 4:
{
  // Switch to high gain linear PD control during Phase 4
  if (inSwinging == 1)
  {
    SendPWM(HIP,PD2PWM(SwingIn,HIP,2,80,8));
  }
  if (inSwinging == 0)
  {
    SendPWM(HIP,PD2PWM(SwingOut,HIP,2,80,8));
  }
  break;
}

}
Appendix C – Dynamic Ankle Push Function

/*****************************************************************************
 * FUNCTION NAME: GetToeOff
 * PURPOSE: Finds toe lift location, scaled based on the current hip position
 * CALLED FROM: ToeLift
 * ARGUMENTS: none
 * RETURNS: int: toeoffPos
*******************************************************************************/
int GetToeOff (void)
{
    int HipState;               // Percentage of full swing, current
    int toeoffPos;             // Desired ToeLift location
    
    HipState = GetHipState();
    if (inback==IN)
    {
        toeoffPos = ((ToeOffIn-PreparedIn)*HipState/100)+PreparedIn;
    }
    else
    {
        toeoffPos = ((ToeOffOut-PreparedOut)*HipState/100)+PreparedOut;
    }
    //printf("InFront: %d, HipState: %d, ToeOff: %d\n", infront, HipState, toeoffPos);
    return toeoffPos;
}

/*****************************************************************************
 * FUNCTION NAME: ToeLift
 * PURPOSE: Determine if toe lift (Sense B) occurs
 * CALLED FROM: State Machine Function
 * ARGUMENTS: none
 * RETURNS: IN, OUT, or NONE
*******************************************************************************/
char ToeLift (void)
{
    char toeoff = NONE;
    int ankle;
    int liftpos;
    ankle = Potentiometer(inback);
    liftpos = GetToeOff();
    if (inback == IN & ankle >= liftpos)
    {
        toeoff = IN;
    }
    else if (inback == OUT & ankle >= liftpos)
    {
        toeoff = OUT;
    }
    return toeoff;
}
Appendix D – Rocking Startup Code

case StateRock:
    // Push on the rear ankles: (RockState = 1)
    // -if zero ankle velocity is reached, then retract rear ankles
    // -wait for the rear ankle to strike the ground again, push with front ankles
    (RockState = -1)
    // -if ToeOff is reached with a positive ankle velocity, then continue on to the walk cycle
    // -goes on to state III
    // Push on the front ankles: (RockState = 2)
    // -if zero ankle velocity is reached, or limiting position is reached, then retract front ankles
    // -wait for the front ankle to strike the ground again, push with rear ankles
    (RockState = -2)
{
    // Return to StateLock (stop trying to go) if the switch if flipped off
    if (GoSwitch == OPEN)
    {
        State = StateLock;
    }

    if (inback == IN)
    {
        // If the hip stance angle is not large enough, turn on hip actuation
        if (GetHipState()<80)
        {
            KpRockHipIn = 0;
        }
        else
        {
            KpRockHipIn = 0;
        }

        // Keep hip stance sufficiently large
        //SendPWM(HIP, PD2PWM(SwingOut, HIP, HipSwingTolerance, KpRockHipIn, KdHipSwing));

        // Push with the rear ankle
        if (RockState == 1)
        {
            // Lock the front ankle
            SendPWM(OUT, PD2PWM(PreparedOut, OUT, AnkleLockToleranceOUT,
            AnkleLockScale, KdPrepareOut));
            // Push rear ankle
            AnklePush();
            printf("\nrockstate = -1\n");
            if (ToeLift[1] == IN & Differentiator(IN)>5)
            {
                State = StateIII; // Assume State III if toelift has been reached
                HipSwingPhase = 1;
                printf("Gone on to the walk cycle\n");
            }
            // Retract the rear ankle, and wait to push again
        else if (RockState == -1)
        {
            // Lock the front ankle
        
            printf("\n\n");
        }
SendPWM(OUT, PD2PWM(PreparedOut, OUT, AnkleLockToleranceOUT, AnkleLockScale, KdPrepareOut));
    // Prepare rear ankle, and wait for rear heelstrike
    SendPWM(IN, PD2PWM(PreparedIn, IN, PrepareToleranceIN, KpPrepareIn, KdPrepareIn));
    // Check for change in RockState
    if (LimitHeelIn == CLOSED)
    {
      RockState = 2;
      printf("rockstate = 2\n");
    }

    // Push with the front ankle
    else if (RockState == 2)
    {
      // Push with front ankle
      SendPWM(OUT, KpFrontAnklePush);
      // Lock rear ankle
      SendPWM(IN, PD2PWM(PreparedIn, IN, AnkleLockToleranceIN, AnkleLockScale, KdPrepareIn));
      // Check for change in RockState
      if (FrontPushStop() == TRUE)
      {
        RockState = -2;
        printf("rockstate = -2\n");
      }
    }

    // Retract the front ankle, and wait to push again
    else if (RockState == -2)
    {
      // Prepare front ankle, and wait for front heelstrike
      SendPWM(OUT, PD2PWM(PreparedOut, OUT, PrepareToleranceOUT, KpPrepareOut, KdPrepareOut));
      // Lock rear ankle
      SendPWM(IN, PD2PWM(PreparedIn, IN, AnkleLockToleranceIN, AnkleLockScale, KdPrepareIn));
      // Check for change in RockState
      if (LimitHeelOut == CLOSED)
      {
        RockState = 1;
        printf("rockstate = 1\n");
      }
    }

    if (inback == OUT) // If the hip stance angle is not large enough, turn on hip actuation
    {
      if (GetHipState() < 80)
      {
        KpRockHipOut = 0;
      }
    }
    else
    {
      KpRockHipOut = 0;
    }

    // Keep hip stance sufficiently large
    // SendPWM(HIP, PD2PWM(SwingIn, HIP, HipSwingTolerance, KpRockHipOut, KdHipSwing));

    // Push with the rear ankle
    if (RockState == 1)
    {
      // Lock the front ankle
      SendPWM(IN, PD2PWM(PreparedIn, IN, AnkleLockToleranceIN, AnkleLockScale, KdPrepareIn));
      // Push rear ankle
      AnklePush();
      // Check for change of RockState
    }
if (Potentiometer(OUT)>(RestPosOut+15) & Differentiator(OUT)<3) {
    RockState = -1; // Stop AnklePush and fall backward if ankle speed becomes too low
}
if (Towelift() == OUT & Differentiator(OUT)>5) {
    State = StateIII; // Assume State III if towelift has been reached
    HipSwingPhase = 1;
    printf("Gone on to the walk cycle");
}

// Retract the rear ankle, and wait to push again
else if (RockState == -1) {
    // Lock the front ankle
    SendPWM(IN, PD2PWM(PreparedIn, IN, AnkleLockToleranceIN, AnkleLockScale, KdPrepareIn));
    // Prepare rear ankle, and wait for rear heelstrike
    SendPWM(OUT, PD2PWM(PreparedOut, OUT, PrepareToleranceOUT, KpPrepareOut, AnkleLockScale, KdPrepareOut));
    // Check for change in RockState
    if (LimitHeelOut == CLOSED) {
        RockState = 2;
    }
}

// Push with the front ankle
else if (RockState == 2) {
    // Push with front ankle
    SendPWM(IN, KpFrontAnklePush);
    // Lock rear ankle
    SendPWM(OUT, PD2PWM(PreparedOut, OUT, AnkleLockToleranceOUT, AnkleLockScale, KdPrepareOut));
    // Check for change in RockState
    if (FrontPushStop() == TRUE) {
        RockState = -2;
    }
}

// Retract the front ankle, and wait to push again
else if (RockState == -2) {
    // Prepare front ankle, and wait for front heelstrike
    SendPWM(IN, PD2PWM(PreparedIn, IN, PrepareToleranceIN, KpPrepareIn, KdPrepareIn));
    // Lock rear ankle
    SendPWM(OUT, PD2PWM(PreparedOut, OUT, AnkleLockToleranceOUT, AnkleLockScale, KdPrepareOut));
    // Check for change in RockState
    if (LimitHeelIn == CLOSED) {
        RockState = 1;
    }
}
break;
Appendix E – Front Foot Push Code

*/FUNCTION NAME: GetStopPos  
* PURPOSE: Finds front push stop location, scaled based on the current hip position  
* CALLED FROM: FrontPushStop  
* ARGUMENTS: none  
* RETURNS: StopPos (int)
*******************************************************************************/
int GetStopPos (void)
{
    int HipState;                                // Percentage of full swing,
    current position
    int PreparedValue;
    long int PushAmount;
    int StopPos;                                // Desired PushStop location

    // Initialize interpolation set points here
    int xPos1 = 50;
    int yPos1;

    // Set the HipState position (percentage of full swing)
    HipState = GetHipState();

    // General form:
    // value = ((y1-y0)*(x-x0))/(x1-x0) + y0

    // Determine vertical position of intermediate points, as well as prepared values
    if (inBack==IN)
    {
        yPos1 = ((PushStopOut-PreparedOut)*50/100)/2;
        PreparedValue = PreparedOut;
    }
    else
    {
        yPos1 = ((PushStopIn-PreparedIn)*50/100)/2;
        PreparedValue = PreparedIn;
    }

    // Perform interpolation based on horizontal position (HipState'), in relation to set
    points
    if (HipState < xPos1)
    {
        PushAmount = ((yPos1-0)*(HipState-0))/(xPos1-0) + 0;
    }
    else
    {
        PushAmount = ((100-yPos1)*(HipState-xPos1))/(100-xPos1) + yPos1;
    }

    // Calculate final position at which front foot push should be stopped
    StopPos = PreparedValue + PushAmount;

    printf("PushStopIn: %d, PushStopOut: %d, HipState: %d\n",PushStopIn,PushStopOut,HipState);
    printf("PushAmount: %d, StopPos: %d\n",PushAmount,StopPos);
    return StopPos;
}

*******************************************************************************/
*/FUNCTION NAME: FrontPushStop  
* PURPOSE: Determine if "front push stop" occurs  
* CALLED FROM: State Machine Function (StateRock)  
* ARGUMENTS: none  
* RETURNS: TRUE or FALSE
*******************************************************************************/
char FrontPushStop (void)
{
char pushStop = FALSE;
int ankle;
int StopPos;
ankle = Potentiometer(infront);

if (infront == IN)
{
    StopPos = GetStopPos();
    if ((Differentiator(infront)<3 & ankle>PreparedIn+15) | (ankle==StopPos))
    {
        pushStop = TRUE;
    }
}
else
{
    StopPos = GetStopPos();
    if ((Differentiator(infront)<3 & ankle>PreparedOut+15) | (ankle==StopPos))
    {
        pushStop = TRUE;
    }
}
return pushStop;
Appendix F – Current Based Control

/******************************************************************************
*FUNCTION NAME: PD2Torque
*PURPOSE: Proportional-Derivative Control function, measures error, converts
* current position/velocity values to recommended torque (current)
*CALLED FROM: Sense Functions
*ARGUMENTS: R, motor, tolerance, Kp, Kd
*RETURNS: void
*******************************************************************************/
void PD2Torque(int R, char motor, char tolerance, int Kp, int Kd)
{
    int position = Potentiometer(motor);
    int velocity = Differentiator(motor);
    int current;

    if (position > R-tolerance & position < R+tolerance)
    {
        // Remove P component if within tolerance
        current = Kd*velocity/10;
    }
    else
    {
        // Linear PD control
        current = (Kp*(R-position))/10 - (Kd*velocity)/10;
    }

    // Limit desired current to max of 7 amps in either direction
    if (current > 390)
    {
        current = 290;
    }
    else if (current < -290)
    {
        current = -290;
    }

    // printf("Target Current: \$d Now Current: \$d\n", current, CurrentSense(IN));
    Torque2PWM(motor, current);
}

/******************************************************************************
*FUNCTION NAME: Torque2PWM
*PURPOSE: Use feedback to set PWM value based on desired torque (current)
*CALLED FROM: Sense Functions
*ARGUMENTS: motor, current
*RETURNS: void
*******************************************************************************/
void Torque2PWM(char motor, int current)
{
    int pwm;
    int pwmLast;
    int CurrentError;
    int CurrentErrorLast;
    int CurrentVelocity;
    int KpCurr = 10;
    int KdCurr = 0;

    // Get previous pwm value
    if (motor==IN)
    {
        pwmLast = pwmLastIn;
        CurrentErrorLast = CurrentErrorLastIn;
    }
    else if (motor==OUT)
    {
        pwmLast = pwmLastOut;
        CurrentErrorLast = CurrentErrorLastOut;
    }
else if (motor==HIP)
{
    pwmLast = pwmLastHip;
    CurrentErrorLast = CurrentErrorLastHip;
} //printf("pwm: %d\n", pwm);

//Set appropriate duty cycle based on desired torque, current torque, and previous duty cycle
if (CurrentSense(motor) < current+4 && CurrentSense(motor) > current-4)
{
    //Keep last pwm value if current is correct
    pwm = pwmLast;
} else
{
    //Current is not correct, so adjust PWM accordingly
    CurrentError = current - CurrentSense(motor);
    CurrentVelocity = Current - CurrentLast;
    pwm = (KpCurr*CurrentError)/10 - (KdCurr*CurrentVelocity)/10;
}

//Save current pwm value for next loop
if (motor==IN)
{
    pwmLastIn = pwm;
    CurrentLastIn = Current;
} else if (motor==OUT)
{
    pwmLastOut = pwm;
    CurrentLastOut = Current;
} else if (motor==HIP)
{
    pwmLastHip = pwm;
    CurrentLastHip = Current;
} //Send pwm value next value
SendPWM(motor, pwm);
## Appendix G – KoBrain Wiring Connectors

<table>
<thead>
<tr>
<th>Innovation First Port</th>
<th>Robot Pin #</th>
<th>Cable Color</th>
<th>Description</th>
<th>KoBrain Port</th>
<th>I/O Block</th>
<th>Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM Out 1</td>
<td>1</td>
<td>Brown</td>
<td>PWM In</td>
<td>PWMA0</td>
<td>SV1</td>
<td>33</td>
</tr>
<tr>
<td>PWM Out 2</td>
<td>2</td>
<td>Red</td>
<td>PWM Out</td>
<td>PWMA1</td>
<td>SV1</td>
<td>34</td>
</tr>
<tr>
<td>PWM Out 3</td>
<td>3</td>
<td>Orange</td>
<td>PWM Hip</td>
<td>PWMA2</td>
<td>SV1</td>
<td>35</td>
</tr>
<tr>
<td>PWM Out 4</td>
<td>4</td>
<td>Yellow</td>
<td></td>
<td>PWMA3</td>
<td>SV1</td>
<td>36</td>
</tr>
<tr>
<td>PWM Out 5</td>
<td>5</td>
<td>Green</td>
<td></td>
<td>PWMB0</td>
<td>SV1</td>
<td>17</td>
</tr>
<tr>
<td>PWM Out 6</td>
<td>6</td>
<td>Blue</td>
<td></td>
<td>PWMB1</td>
<td>SV1</td>
<td>18</td>
</tr>
<tr>
<td>PWM Out 7</td>
<td>7</td>
<td>Purple</td>
<td></td>
<td>PWMB2</td>
<td>SV1</td>
<td>19</td>
</tr>
<tr>
<td>PWM Out 8</td>
<td>8</td>
<td>Grey</td>
<td></td>
<td>PWMB3</td>
<td>SV1</td>
<td>20</td>
</tr>
<tr>
<td>Digital/Analog In 1</td>
<td>9</td>
<td>White</td>
<td>Inner pot signal</td>
<td>AN0</td>
<td>AN 1/2</td>
<td>1</td>
</tr>
<tr>
<td>Digital/Analog In 2</td>
<td>10</td>
<td>Black</td>
<td>Outer pot signal</td>
<td>AN1</td>
<td>AN 1/2</td>
<td>2</td>
</tr>
<tr>
<td>Digital/Analog In 3</td>
<td>11</td>
<td>Brown</td>
<td>Hip pot signal</td>
<td>AN2</td>
<td>AN 1/2</td>
<td>3</td>
</tr>
<tr>
<td>Digital/Analog In 4</td>
<td>12</td>
<td>Red</td>
<td>Inner diff signal</td>
<td>AN3</td>
<td>AN 1/2</td>
<td>4</td>
</tr>
<tr>
<td>Digital/Analog In 5</td>
<td>13</td>
<td>Orange</td>
<td>Outer diff signal</td>
<td>AN4</td>
<td>AN 1/2</td>
<td>5</td>
</tr>
<tr>
<td>Digital/Analog In 6</td>
<td>14</td>
<td>Yellow</td>
<td>Hip diff signal</td>
<td>AN5</td>
<td>AN 1/2</td>
<td>6</td>
</tr>
<tr>
<td>Digital/Analog In 7</td>
<td>15</td>
<td>Green</td>
<td>Hip current</td>
<td>AN6</td>
<td>AN 1/2</td>
<td>7</td>
</tr>
<tr>
<td>Digital/Analog In 8</td>
<td>16</td>
<td>Blue</td>
<td>Out current</td>
<td>AN7</td>
<td>AN 1/2</td>
<td>8</td>
</tr>
<tr>
<td>Digital/Analog In 9</td>
<td>17</td>
<td>Purple</td>
<td>In current</td>
<td>AN8</td>
<td>AN 1/2</td>
<td>9</td>
</tr>
<tr>
<td>Digital/Analog In 10</td>
<td>18</td>
<td>Grey</td>
<td>Battery voltage</td>
<td>AN9</td>
<td>AN 1/2</td>
<td>10</td>
</tr>
<tr>
<td>Digital/Analog In 11</td>
<td>19</td>
<td>White</td>
<td>Gyros</td>
<td>AN10</td>
<td>AN 1/2</td>
<td>11</td>
</tr>
<tr>
<td>Digital/Analog In 12</td>
<td>20</td>
<td>Black</td>
<td>Gyros</td>
<td>AN11</td>
<td>AN 1/2</td>
<td>12</td>
</tr>
<tr>
<td>Digital/Analog In 13</td>
<td>21</td>
<td>Brown</td>
<td></td>
<td>AN12</td>
<td>AN 1/2</td>
<td>13</td>
</tr>
<tr>
<td>Digital/Analog In 14</td>
<td>22</td>
<td>Red</td>
<td>Motor direction IN</td>
<td>GPIOB0</td>
<td>SV1</td>
<td>9</td>
</tr>
<tr>
<td>Digital/Analog In 15</td>
<td>23</td>
<td>Orange</td>
<td>Motor direction OUT</td>
<td>GPIOB1</td>
<td>SV1</td>
<td>10</td>
</tr>
<tr>
<td>Digital/Analog In 16</td>
<td>24</td>
<td>Yellow</td>
<td>Motor direction HIP</td>
<td>GPIOB2</td>
<td>SV1</td>
<td>11</td>
</tr>
<tr>
<td>GND DOAI 12</td>
<td>25</td>
<td>Green</td>
<td>GND</td>
<td>GND</td>
<td>SV1</td>
<td>39</td>
</tr>
<tr>
<td>GND DOAI 13</td>
<td>26</td>
<td>Blue</td>
<td>GND</td>
<td>GND</td>
<td>SV1</td>
<td>40</td>
</tr>
<tr>
<td>GND DOAI 14</td>
<td>27</td>
<td>Purple</td>
<td>GND</td>
<td>GND</td>
<td>SV1</td>
<td></td>
</tr>
<tr>
<td>GND DOAI 15</td>
<td>28</td>
<td>Grey</td>
<td>GND</td>
<td>GND</td>
<td>SV1</td>
<td></td>
</tr>
<tr>
<td>GND DOAI 16</td>
<td>29</td>
<td>White</td>
<td>GND</td>
<td>GND</td>
<td>SV1</td>
<td></td>
</tr>
<tr>
<td>+5V DOAI 12</td>
<td>30</td>
<td>Black</td>
<td>+5V</td>
<td>5V</td>
<td>SV12</td>
<td>9</td>
</tr>
<tr>
<td>+5V DOAI 13</td>
<td>31</td>
<td>Brown</td>
<td>+5V</td>
<td>5V</td>
<td>SV12</td>
<td>10</td>
</tr>
<tr>
<td>+5V DOAI 14</td>
<td>32</td>
<td>Red</td>
<td>+5V</td>
<td>5V</td>
<td>SV12</td>
<td></td>
</tr>
<tr>
<td>+5V DOAI 15</td>
<td>33</td>
<td>Orange</td>
<td>+5V</td>
<td>5V</td>
<td>SV12</td>
<td></td>
</tr>
<tr>
<td>+5V DOAI 16</td>
<td>34</td>
<td>Yellow</td>
<td>+5V</td>
<td>5V</td>
<td>SV12</td>
<td></td>
</tr>
<tr>
<td>Interrupt 1</td>
<td>35</td>
<td>Green</td>
<td>Inner limit switch</td>
<td>GPIOA0</td>
<td>SV1</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt 2</td>
<td>36</td>
<td>Blue</td>
<td>Outer limit switch</td>
<td>GPIOA1</td>
<td>SV1</td>
<td>2</td>
</tr>
<tr>
<td>Interrupt 3</td>
<td>37</td>
<td>Purple</td>
<td>Calibration button</td>
<td>GPIOA2</td>
<td>SV1</td>
<td>3</td>
</tr>
<tr>
<td>Interrupt 4</td>
<td>38</td>
<td>Grey</td>
<td>Start/stop switch</td>
<td>GPIOA3</td>
<td>SV1</td>
<td>4</td>
</tr>
<tr>
<td>Interrupt 5</td>
<td>39</td>
<td>White</td>
<td></td>
<td>GPIOA4</td>
<td>SV1</td>
<td>5</td>
</tr>
<tr>
<td>Interrupt 6</td>
<td>40</td>
<td>Black</td>
<td></td>
<td>GPIOA5</td>
<td>SV1</td>
<td>6</td>
</tr>
<tr>
<td>Robot Pin #</td>
<td>Cable Color</td>
<td>Description</td>
<td>KoBrain Port</td>
<td>I/O Block</td>
<td>Pin</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------------</td>
<td>--------------</td>
<td>-----------</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Brown</td>
<td>HIP CH A (3)</td>
<td>GPIOE13</td>
<td>SV2</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>HIP CH B (5)</td>
<td>GPIOE12</td>
<td>SV2</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Orange</td>
<td>OUT CH A (3)</td>
<td>GPIOE11</td>
<td>SV2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Yellow</td>
<td>OUT CH B (5)</td>
<td>GPIOE10</td>
<td>SV2</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Green</td>
<td>IN CH A (3)</td>
<td>GPIOE9</td>
<td>SV2</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Blue</td>
<td>IN CH B (5)</td>
<td>GPIOE8</td>
<td>SV2</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Purple</td>
<td>GND</td>
<td>GND</td>
<td>SV12</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Grey</td>
<td>3.3V</td>
<td>3.3V</td>
<td>SV12</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>White</td>
<td>GND</td>
<td>GND</td>
<td>SV12</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Black</td>
<td>3.3V</td>
<td>3.3V</td>
<td>SV12</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>