ABSTRACT

We have created a simulated environment in Matlab that models a 2D 2-link brachiating robot using Lagrangian dynamics. We then developed a dynamic programming based control scheme to allow the robot to perform swing-up and continuous contact brachiating swing.

INTRODUCTION

Branching structures are a ubiquitous element in several environments on earth, from trees found in nature to man-made trusses and power lines. Being able to navigate such environments provides a difficult challenge to robots ill-equipped to handle the task. In nature, movement through such an environment is solved by apes through a process called brachiation, where locomotion through a forest canopy is done by hand-over-hand swinging. Gibbons, a type of ape from Southeast Asia, have adopted this as their primary form of locomotion, ranging from a pace equivalent to human walking speeds all the way up to astonishing speeds of 35m/h when startled [1]. If it is possible to replicate this method of locomotion in a robot, it would offer desirable advantages including an effective way to gain a higher vantage point and the opportunity to perform desirable tasks in such environments such as sample collection of arboreal flora/fauna, exploration, or power line inspections. Unfortunately, brachiation is a complex maneuver that is under-actuated, multivariate, hard to optimize, and needs to be fast. Therefore, controlling for brachiation is non-trivial. In order to simplify the problem, the type of brachiation examined in this work is called a continuous contact brachiating gait, because at least one arm is in contact with the environment at all times. This is a subset of the more general but more complex ricochetal brachiation gait, where there are ballistic flight phases.

Brachiation was first compared to that of a pendulum in the 1960s [2]. Fukuda et al. also examined approaches considering the dynamics of the system and using plant-inversion techniques [3]. This however, requires precise mathematical models of the robot. Hasegawa and Fukuda also proposed a behavior-based controller to reduce the number of degrees of freedom to be considered [4]. Their approach was to decompose behaviors into a hierarchical structure and then to co-ordinate them together to perform swing motions. This approach requires accurate tuning each time the brachiator configuration is modified. Other approaches include model-based predictive controls [5], where the motion of the brachiator is modeled and simplifications are made to linearize and solve the dynamic model, which works for simple brachiators but may not be ideal for complicated multi-link brachiators. Another approach is to use evolutionary algorithms [6], where energy efficient brachiation swing motions can be developed. This method has the drawback of not being real-time and is only tuned for the particular simulated environment that must be observed in advance. There are also works in examining passive-swing dynamics [7], where swing-motions without the need of actuators are examined, but such motions are restricted to frictionless swing motions with only small variances in the branch’s horizontal position, and no variation in the branch’s vertical position as we cannot swing up without energy injection.

For a two-link brachiator, we can also look at previous works on double pendulums for inspiration on how to solve the problem. Brachiation can be split into swing-up and the actual brachiation swing, which are both a specific subset of the double-pendulum swing-up problem. One of the approaches used previously is to use dynamic programming to solve this problem [8]. This solution is desirable because we can train the brachiator offline without having to work with the modeling needed by the previous approaches. The algorithm will do the heavy lifting involved in designing the swing strategy.

MODEL

The two-link robot model for the brachiating robot considered in this work is shown in Fig. 1. By modeling the brachiating robot as an open chain manipulator, its dynamics can be given by the Lagrange equation:

\[ M(\theta) \ddot{\theta} + C(\theta, \dot{\theta}) \dot{\theta} + N(\theta, \dot{\theta}) = \tau \]  

(1)

Here, \( M(\theta) \) is the inertia matrix. \( C(\theta, \dot{\theta}) \) is the Coriolis matrix, where the vector \( C(\theta, \dot{\theta}) \dot{\theta} \) contains the Coriolis and centrifugal force terms. \( N(\theta, \dot{\theta}) \) includes the gravity
forces and friction which acts on the joints. Finally, $\tau$ is the actuator torques. Computation of these terms is covered in Appendix A.

Figure 1: The 2-link brachiator.

With the equations of motion computed, we can solve for the angular acceleration $\dot{\theta}(t)$ at each time step by plugging in our current angular position and velocities and our applied torques and then using Matlab’s symbolic solver. We then integrate the angular acceleration to get the changes in velocity and position at that time step.

A 1ms time step was selected as a tradeoff between simulation speed and accuracy. Due to the time step and the simple integration strategy used, there were some errors between the Matlab simulation in comparison to the more accurate Simulink model. Figure 2 shows a comparison between a zero torque and zero friction swing from a horizontal starting position using the Matlab model and a Simulink simulation. Note that the Matlab model starts to lose total energy due to numerical errors, and would diverge from the Simulink output due to the chaotic nature of the motion at around 5s. The Matlab simulation loses roughly 2J of energy every 10s, or 0.2J per second. Since brachiation might have swings of up to 20J per second, this error only represents only 1% of error. Using the Matlab simulation was desirable because the equations of motion were easily portable into a C++ code for training the dynamic programming tables at a high speed, whereas the more accurate Simulink method was slower and only used for verification after training.

Figure 3: A comparison between the Matlab and Simulink simulations with zero friction starting from a straight down position and with a constant torque of 5Nm. Note that the energy is not constant in this system as we are only looking at the gravitational potential energy and kinetic energies, not the stored spring energy caused by the constant torque.

Figure 4 shows two simulations of random torques sampled at 20ms intervals ranging from -10 to 10Nm applied to the brachiator from neutral position. The same random torques are applied to both the Matlab and Simulink simulations. Note that due to the chaotic nature of the system and small differences in how the Matlab simulation computes the integration versus the Simulink simulation, the energies start to diverge after 5s of simulation. Some of these differences include the existence of an ode approximation in the Simulink simulation that was not present in the Matlab simulation. Other differences include the order of summation of the acceleration, velocity, and position terms, and when the torque was updated with respect to computing the dynamics.

The Matlab simulation is treated as the gold standard, but the Matlab simulation was easier to develop for and was simple to convert to a C++ version for faster computation. Since the two simulations are approximately
equal up to 5s, using the Matlab/C++ version was considered adequate for our experiments. Note that all simulations would eventually diverge from a real world example due to modeling errors.

We seek to use dynamic programming to design a swing controller. We start by converting the continuous time and continuous state brachiation problem into one that has discrete time and discrete states. The state and following state was represented by:

\[ X = (\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2) \]

\[ X_{k+1} = f(X_k, u) \]

The time slice selected was 0.02s, which was slower than the simulation time of 0.001s. This was because a large enough time step was needed between actions or else there would be no state change per action due to the coarse state discretization. Therefore, we query the controller for a new torques every 20ms while simulating at 1ms intervals. Figure 5 shows the effect of a larger time slice.

\[ \Delta t = 1ms \quad \text{and} \quad \Delta t = 20ns \]

\[ X_{k+1} \]

The state slice selected was 0.02s, which was slower than the simulation time of 0.001s. This was because a large enough time step was needed between actions or else there would be no state change per action due to the coarse state discretization. Therefore, we query the controller for a new torques every 20ms while simulating at 1ms intervals. Figure 5 shows the effect of a larger time slice.

Figure 5: A comparison between a smaller time slice and a larger one. If the time slice is too small, an action may not cause a change in state if the state quantization is too large.

The state resolution selected was 100x100x100x100, as recommended by Atkeson et al who found that lower state-resolutions failed to reveal solutions for two-link systems with full control on both joints [8]. Note that our control problem is slightly different in that we do not have full control of the system due to lacking actuation at the hand joint. The state resolution represented 100 million states per branch location, and required 100 million more states for each additional branch location. The target branch locations were selected from between 40-60cm from the current branch, at 5cm intervals. The action represented the elbow torque for each state, and the action space was continuous between the minimum and maximum allowed torque of 10Nm:

\[ u(X) \in \mathbb{R}, -10 < u(X) < 10 \]

The cost function selected was defined as:

\[ L(X, u) = (p - p_{desired})^2 \]

The end effector position \( p \) was found using the forward kinematics. The Q function used was defined as:

\[ Q(X, u) = L(X, u) + \gamma V(f(X, u)), 0 < \gamma < 1 \]

\( \gamma \) represents the weighting factor between immediate and future rewards. If \( \gamma \) is closer to zero, the controller would select for immediate rewards, while if it is closer to one the controller would select for future rewards. We set \( \gamma = 1 \) because during swing up, the robot might need to move away from the goal in order to swing back and forth and build momentum, and therefore we would want the controller to focus on the future rewards rather than immediate rewards. Finally, the Bellman update equation for the value at each state was defined as:

\[ V(X) = \min_u Q(X, u) \]

In typical dynamic programming training, the naïve trainer would provide a sample of possible actions for each of the 100 million states and evaluate the Bellman update equation to generate the value table for the next step. However, the time needed to produce enough value table updates for the state space we have is too large, and therefore a better training algorithm is needed. Atkeson et al found that random action dynamic programming is fast and effective for training the value table [8]. This algorithm is outlined in Figure 6. Convergence is defined as when \( V(X) \) no longer changes, however in our implementation it was found that stopping at 1000 sweeps of random actions for each state was sufficient. At 1000 sweeps, less than 0.001% of the states are updated with new actions per sweep.

The training time required per sweep of all the states was approximately 28s when implemented in C++ on an i7-3970x processor. Therefore, for 1000 sweeps the algorithm needed 8h per new branch position. For our 5 branch positions, which were located at 40, 45, 50, 55, and 60cm from the current branch with no change in y-position, this algorithm took 40h to complete. For every possible branch location, the policy \( u(X) \) required a table with 100 million double values, representing roughly 0.8GB per table in C++. Therefore, storing the policy for 5 branch positions requires 4GB of memory, and in an real-world scenario more branch locations would be needed requiring even more memory.

Figure 7 shows a slice of the value and action space that was trained using this method. The slice is the tables at zero angular velocities at both joints. A state of 50 represents zero angular position, where zero on both angles is straight up. There are two areas of lowest value, representing the elbow-up and elbow-down positions for grasping the target branch. Note that around a joint angle of zero for the elbow represented as a theta 2 state of 7-3970x processor. Therefore, for 1000 sweeps the algorithm needed 8h per new branch position. For our 5 branch positions, which were located at 40, 45, 50, 55, and 60cm from the current branch with no change in y-position, this algorithm took 40h to complete. For every possible branch location, the policy \( u(X) \) required a table with 100 million double values, representing roughly 0.8GB per table in C++. Therefore, storing the policy for 5 branch positions requires 4GB of memory, and in an real-world scenario more branch locations would be needed requiring even more memory.

Figure 7 shows a slice of the value and action space that was trained using this method. The slice is the tables at zero angular velocities at both joints. A state of 50 represents zero angular position, where zero on both angles is straight up. There are two areas of lowest value, representing the elbow-up and elbow-down positions for grasping the target branch. Note that around a joint angle of zero for the elbow represented as a theta 2 state of 7-3970x processor. Therefore, for 1000 sweeps the algorithm needed 8h per new branch position. For our 5 branch positions, which were located at 40, 45, 50, 55, and 60cm from the current branch with no change in y-position, this algorithm took 40h to complete. For every possible branch location, the policy \( u(X) \) required a table with 100 million double values, representing roughly 0.8GB per table in C++. Therefore, storing the policy for 5 branch positions requires 4GB of memory, and in an real-world scenario more branch locations would be needed requiring even more memory.

Figure 7 shows a slice of the value and action space that was trained using this method. The slice is the tables at zero angular velocities at both joints. A state of 50 represents zero angular position, where zero on both angles is straight up. There are two areas of lowest value, representing the elbow-up and elbow-down positions for grasping the target branch. Note that around a joint angle of zero for the elbow represented as a theta 2 state of between 40-60, the torques are quite noisy compared to the flat planes outside this range. This indicates that there may be additional dynamics in this area that are not entirely captured by our coarse state grid.
RESULTS

Swing-Up

Figure 8 shows the both joint angles of a swing-up from a straight down hanging position attempting to grasp a branch located 56.8cm away from the current position. The control policy selected is whichever trained target branch policy is closest to this actual branch location. In this case, the controller selects the policy trained with a target branch distance of 55cm. Note that there was chattering during the first 4s. This was due to the dynamics of the swing motion around the manipulator singularity of being perfectly straight being more detailed than could be described by the state table due to the quantization into discrete values. Once it moved further from this position, it was able to perform swing up until it grasped the target branch. This chattering may be removed in future controllers with a coarser state grid near states where $\theta_2 = 0$, although the already large training time prevented more states from being added when preparing this paper.

At roughly 5.5s, the controller decided to rotate the elbow past the arm, performing an overhead swing. This is indicated by the transition of the joint angle from -180 degrees to +180 degrees. Although this was valid in simulation, joint limits must be added before applying this to the real world as the physical robot cannot rotate the elbow past the arm. Figure 9 shows both joint velocities of the same swing-up motion. There are large spikes in joint velocities as the robot snaps back and forth from the vertical singularity position during the first 4s. Since the controller was not trained to optimize for soft landings, there is still considerable angular velocity in the elbow joint as the robot makes contact with the target branch.
Figure 9: The angular velocity of the swing-up motion. The controller does not optimize for soft landings, so there is a large elbow velocity when it reaches the branch at 5.5s.

Figure 10 shows the energies of the swing-up motion. Note that very little energy was stored during chattering in the first 4s. Once it moved beyond the vertical position, energy accumulation begins. When the robot made the grasp, the robot had roughly the same potential energy as the minimum required energy, as expected. Any differences were caused by the target bar having a non-zero diameter, which on contact with the end effector stopped the simulation. Therefore, the end effector did not need to hit the target point dead on, but had 2.5cm of leeway from the center. Finally, note that on impact there was considerable excess kinetic energy, leading to a large overshoot in total energy of roughly 15J. This would cause an undesirable impact force in a real system.

Figure 11: The torque input on the elbow joint for swing-up. It is extremely detailed and would be hard to implement in the physical robot.

The swing-up was successful, however we required a joint limit on the elbow joint that prevented it from rotating past the arm. Adding this joint limit produced a different swing-up motion. Figure 12 and Figure 13 shows the angular position and velocities respectively. There is still 4s of chattering as before, but the actual swing-up takes more time, grasping at 8s instead of 5.5s. However, there were no transitions of the elbow joint from -180 degrees to +180 degrees, as desired. Unfortunately, the final joint velocities were again non-zero, and were actually larger in this swing-up with a large hand angular velocity on impact.

Figure 12: The angular position of a swing-up with joint limits that prevented the elbow from rotating past the arm. The 4s of chattering remained as in the previous swing. The swing-up took longer than in the swing-up without joint limits, taking 8s instead of 5.5s.

Figure 13: The angular velocity of the swing-up with the joint limits. The joint velocities on impact at 8s were non-zero.
Figure 14 and Figure 15 show the energies and torques respectively of the swing-up with joint limits. As before, the potential energy is roughly the same as the minimum required energy, with differences caused by the target bar having a diameter. Again, there was undesirable excess energy on impact, and the torque is extremely detailed and not suitable for the real system. Note that the swing energy is identical to the swing energy as before until 5.5s where the previous controller was able to make the grasp. Due to the joint limits, the motion was hindered and energy was lost, requiring additional swings and time to perform the grasp.

Brachiation

Since we are performing continuous contact brachiating gait of a two-link brachiator, we only need to demonstrate a single brachiation swing per iteration of the previous and next branch positions. If this is possible, then every subsequent swing is simply a repeat of the demonstrated swings. Figure 16 and Figure 17 show the brachiation swing using a ladder separation of 55cm. Note that the initial joint angles is an inverse of the final joint angles, with any differences caused by the target branch having a radius of 2.5cm, and the simulation ending on contact with the edge of this circle instead of the center.

Figure 16: The angular position of a single brachiation swing. Note that the initial joint angles mirror the final joint angles as expected. Any difference is caused by the target branch having non-zero diameter.

Figure 17: The angular velocity of a single brachiation swing.

Figure 18 and Figure 19 show the energy and torque of the swing respectively. Note that the final potential energy is approximately equal to the minimum required energy as expected. There was an overshoot in total energy as the kinetic energy on contact with the target branch was non-zero.

Figure 18: The energies of a single brachiation swing. The final potential energy was roughly the same as the final required energy as expected. There was undesirable excess kinetic energy on impact.
We can also improve attributes of the swing by imposing a torque penalty to the cost function in training. Figure 20 and Figure 21 show the energies and cost for a brachiation swing when torque penalty was applied. Note that there is a reduction in torque, and the overshoot in energy was reduced. By imposing the penalty, the average magnitude of the torque was reduced from 5.9851Nm to 3.3337Nm for a brachiation swing. Although torques do not directly relate to energy, a DC motor does require current to produce a torque. Therefore, smaller average torques mean less current overall. Additionally, the excess kinetic energy on impact was reduced from 7J to 2.5J.

**CONCLUSION**

We have shown that dynamic programming can be used to train for a brachiation controller to solve for both swing-up and brachiation swings. The training time was found to be costly, but it is offline. Running the actual algorithm is fast since it is a simple lookup table. We have also shown that we can improve performance by altering the cost function to penalize for torques.

Future works might find it beneficial to explore trajectory-based dynamic programming strategies to better perform brachiation and to extend this strategy to brachiators with more body segments. A bootstrap should also be looked at to help avoid chattering on startup. Finally, more advanced cost functions should be explored, such as one that penalizes for any excess energy during swinging or excess impact energy for softer landings.

**REFERENCES**


APPENDIX A – Solving the Equations of Motion

For simplification and without loss of generality, we have set the manipulator lengths to be \( l_i = 1 \) and the center of mass of each linkage to be at the center of that linkage. Since our manipulator is in 2D, the rotations of our joints are in the \( z \)-direction:

\[
\omega_i = \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

Our joint positions at rest are:

\[
q_1 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},
q_2 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}
\]

The joint twists are given by:

\[
\xi_i = \left[ \begin{array}{c} v_i \\ \omega_i \end{array} \right], \quad v_i = -\omega_i \times q_i
\]

To each link we attach a frame \( L_i \) at the center of mass and aligned with the principle inertia axes of the link:

\[
g_{st_1(0)} = \begin{bmatrix} I \\ 0.5 \\ 0 \\ 0 \\ 1 \\ 0.5 \\ 0 \end{bmatrix},
g_{st_2(0)} = \begin{bmatrix} I \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}
\]

With this choice of link frames, the link inertia matrices are:

\[
M_i = \begin{bmatrix} I * m_i \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}
\]

We have set \( m_i = 1 \), \( L_{xi} = \frac{m_i l_i^2}{12}, \) and \( L_{yi} = L_{zi} = 0 \).

In addition, we need:

\[
e^{\xi_i \theta_i} \left( I - e^{\xi_i \theta_i} (\omega_i \times v_i) + \omega_i \omega_i^T v_i \theta_i \right)
\]

Where:

\[
e^{\omega_i \theta_i} = I + \omega_i \sin \theta_i + \tilde{\omega}_i \tilde{\omega}_i (1 - \cos \theta_i),
\]

\[
\tilde{\omega} = \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}
\]

We also need the Jacobian:

\[
J_i(\theta) = \left[ \begin{array}{c} \xi_i^T \\ \xi_{i-1}^T \\ \xi_i^T \\ 0_i to n \end{array} \right],
\]

\[
\xi_j^+ = Ad_{e^{\xi_j \theta_j} g_{st_j(0)}}^{-1} e^{\xi_j \theta_j} g_{st_j(0)}
\]

Where:

\[
e^{\xi_j \theta_j} g_{st_j(0)} = \begin{bmatrix} R & p \\ 0 & 1 \end{bmatrix},
Ad_{e^{\xi_j \theta_j} g_{st_j(0)}}^{-1} = \begin{bmatrix} R^T & -R^T p \\ 0 & R \end{bmatrix}
\]

\[
\hat{p} = \begin{bmatrix} 0 & -p_3 & p_2 \\ p_3 & 0 & -p_1 \\ p_2 & p_1 & 0 \end{bmatrix}
\]

Therefore, we have:

\[
J_1(\theta) = \begin{bmatrix} 0 & 0 \end{bmatrix}, J_2(\theta) = \begin{bmatrix} \sin \theta_2 & 0 \\ \cos \theta_2 + 0.5 & 0.5 \end{bmatrix}
\]

For our inertial matrix, we have:

\[
M(\theta) = \sum_{i=1}^{n} J_i^T(\theta)M_i J_i(\theta)
\]

And from our inertial matrix we can get our Coriolis matrix:

\[
C_{ij}(\theta, \dot{\theta}) = \frac{1}{2} \sum_{k=1}^{n} \left( \frac{\partial M_{ij}(\theta)}{\partial \theta_k} + \frac{\partial M_{ik}(\theta)}{\partial \theta_j} - \frac{\partial M_{kj}(\theta)}{\partial \theta_i} \right) \dot{\theta}_k
\]

Finally, we need:

\[
N(\theta, \dot{\theta}) = \begin{bmatrix} \frac{\partial E_p(\theta)}{\partial \theta} \\ \frac{\partial E_p(\theta)}{\partial \theta} \end{bmatrix}
\]

Where \( E_p \) is the potential energy of the manipulator:

\[
E_p(\theta) = \sum_{i=1}^{n} m_i g h_i(\theta),
\]

\[
h_i(\theta) = e^{\xi_i \theta_i} g_{st_i(0)}(2,4)
\]

As a pair to the potential energy, the Kinetic energy is given by:

\[
E_k(\theta) = \frac{1}{2} \dot{\theta}^T M(\theta) \dot{\theta}
\]