DYNAMIC PROGRAMMING OF A TORSO ACTUATED RIMLESS WHEEL ROBOT

by

ROBERT BROTHERS, B.Sc.

THESIS
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

COMMITTEE MEMBERS:
Pranav Bhounsule, Ph.D., Chair
Robert Hood, Ph.D.
Hung Da Wan, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Mechanical Engineering
December 2018

Copyright 2018 Robert Brothers
All rights reserved.

DEDICATION

I would like to dedicate this thesis to my Mother and Father, for they are the giants whose shoulders I stand upon.

ACKNOWLEDGEMENTS

First of all, I would like to thank Pranav Bhounsule for his guidance and leadership through this experience and for the opportunities presented. I would also like to thank Dr. Robert Lyle Hood and Dr. Hung Da Wan for their time and attentions by being member of this thesis' committee. In addition I would like to thank Sebastian Sanchez, Ezra Ameperosa, Rico Ulep, Scott Miller, and Kyle Seale for their past/current work in the area of legged robotics and their contribution to this thesis via designed and tested prototype on which this thesis builds. I would also like to extend thanks to Ali Zamani, Dr. Ahmad Taha, George Brothers for their support and all who reviewed and took part in the production of this document.

December 2018

DYNAMIC PROGRAMMING OF A TORSO ACTUATED RIMLESS WHEEL ROBOT

Robert Brothers, M.Sc.
The University of Texas at San Antonio, 2018

Supervising Professor: Pranav Bhounsule, Ph.D.

This thesis presents the variable speed walking of a rimless wheel robot. The rimless wheel is a dynamic walking template that consists of a wheel whose rim has been removed (a discrete wheel). By swinging the torso forward beyond the contact point of the leading foot with respect to the gravity vector, the robot is able to move forward. We use Dynamic Programming for optimizing the energy usage when achieving the speed transition. One issue, however, is that the accuracy of the Dynamic Programming solution depends on the fineness of the grid. This is particularly problematic for systems with high degrees of freedom, where a fine state-space discretization will lead to exponential growth in the computation and storage. To circumvent this issue, we use the step-to-step map, also known as the Poincaré map to discretize the system. The Poincaré map relates the velocity of the rimless wheel robot at the mid-stance (the state) and the fixed torso angle per step (the control variable) to the velocity at mid-stance of the next step (the new state). Using this map, we set up a Dynamic Programming problem to minimize a weighted sum of the normalized squared deviation from the desired speed and the normalized squared actuator effort. The problem is then solved using a combination of value- and policy- iteration for computational efficiency. We demonstrate that it is possible to switch from a mid-stance speed of 2 $\frac{rad}{s}$ to 3$\frac{rad}{s}$ in 6 steps. Our results suggest that Poincaré map based Dynamic Programming is a computationally efficient paradigm for solving high dimensional problems in legged locomotion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

This thesis presents a deliberation in the application of Dynamic Programming for the variable speed control of a simplified model of the Rowdy Runner II, a torso actuated rimless wheel robot. Rowdy consists of a torso sandwiched between two rimless wheels. By applying a torque between the torso and wheels, forward motion is produced when the torso's center of mass protrudes beyond the contact point of the leading foot with respect to the gravity vector. Currently, the Rowdy Runner II is capable of controlling the angle of the torso during motion to maintain a constant velocity. The work presented in this thesis uses Dynamic Programming for optimizing the energy usage for achieving the speed transition. One issue, however, is that the accuracy of the Dynamic Programming solution depends on the resolution of the grid. In practice, Dynamic Programming is known to require disproportionally larger amounts of computational time as the complexity of a problem grows, a relation known as the Curse of Dimensionality. This creates a dilemma: on the one hand, a fine grid is essential for good controller performance while on the other, finding these optimal policies at higher resolution require a rather large amount of time. To circumvent this issue, a step-to-step map, also known as the Poincaré map, is used to discretize system's state. The Poincaré map relates the mid-stance velocity of the rimless wheel (the state) and the fixed torso angle per step (the control variable) to the mid-stance velocity at the next step (the new state). Using this map, a Dynamic Programming problem to minimize a weighted sum of the square of the deviation from the desired speed and the actuator effort is set up. This approach reduces the number of states to consider, simplifying the Dynamic Programming problem and reducing computational time. The optimal policies for speed transitions are then found using a combination of policy- and value- iteration for computational efficiency. The results of this work show that it is possible to switch from a mid-stance velocity of 2 $\frac{rad}{s}$ to 3 $\frac{rad}{s}$ in less than 6 steps.

1.1 Introduction to Dynamic Programming

Dynamic Programming is a set of mathematical tools used to reduce an optimization problem to a set of sequential decisions and, from them, determine optimal policies. First proposed by Richard Bellman, Dynamic Programming applies his Principle of Optimality in combination with Markovian Decision Processes to recursively extract optimal policies from nested problems i.e. problems within problems. To better understand the process of Dynamic Programming and eventually devise a method for its implementation the traditional approaches to generating and solving Dynamic Programming are evaluated in a "shortest path" example of a man optimizing his route to work [1].

1.1.1 Example: Man Biking to Work

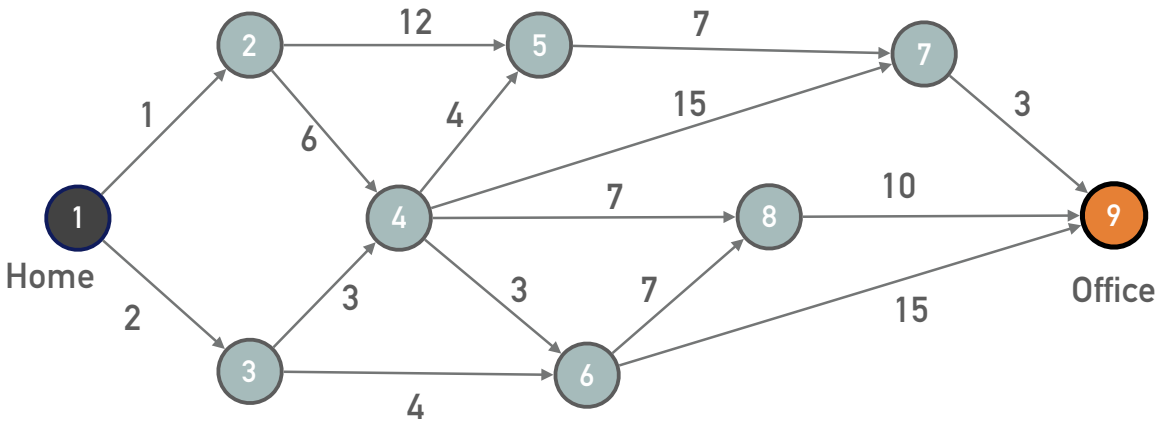

Figure 1.1: Nodal network of man biking to work

Each day, John travels to work on his bicycle. With some exploration each day, he realizes there are a number of different routes that get him there, each requiring different amounts of time. He would, of course, like to find the route that will get him to work the in the shortest amount of time. For this, he spends some time evaluating different paths, and after recording information about all the available paths to work, he organizes a network of nodes interconnected by arcs. In this network, arcs represent streets while the nodes, represent the intersections of streets. Next to each arc (street) he denotes the amount of time required to travel the street and reach the next node. Assuming he can only travel forward, he drafts Figure 1.1 depicting a network of routes to work

available to John. From it, the objective is to find the fastest route or, temporally, the shortest path. Traditionally there are 2 methods used find the shortest path for a network of this form: value- and policy- iteration. These iterative methods are known as Markov Decision Processes.

Value Iteration

Value iteration is a decision making process where the optimal policy is found by searching the network for the optimal value function [2, 3]. This process works as described in Algorithm 1.1

Algorithm 1.1 Value iteration algorithm

```
Initialize Value Function
while ϵ ≤ V(s) − min_a Q(s, a) do
  for all s ∈ S do
    for all a ∈ A do
      Q(s, a) := J(s, a) + V(s')
    V(s) := min_a Q(s, a)
```

[4]. First initialize the value function at the goal state to zero. This value function is a Bellman Equation and for deterministic systems, is expressed as shown in Equation 1.1. Literally this equation translates to: the value of transitioning from state *s* to the goal state is equal to the sum of the expected cost $J(s, a)$ accrued by applying action *a* to transition to state *s'* and the value $V(s')$ at *s'*. At the goal state because there is no subsequent state there is no transition cost therefore the value function at the goal state is equal to zero.

$$V(s) = J(s, a) + V(s') \tag{1.1}$$

Begin at some state *s*, use Equation 1.1 to express the value gained by taking action a. If the value $V(s')$ is known calculate the solution, otherwise evaluate the value function $V(s')$ to find $V(s') = V(s', a, V(s''))$ i.e. that the value function $V(s')$ is a function of the state s', the action a, and the value function of the subsequent state $V(s'')$. Repeatedly perform this operation this until arriving a value function whose $V(s^i)$ has a numerical solution. Then cascading backwards until reaching the nucleate value function $V(s)$ solve the value functions. An iteration is considered

complete when the all the states in the network have been evaluated. This approach, though straight forward, requires a large amount of working memory.

Policy Iteration

Another method of finding the fastest route to work is using policy iteration. Policy iteration is a decision making process that finds the optimal path by directly manipulating the policy [3, 5]. The

Algorithm 1.2 Policy iteration algorithm

```
1: Choose an Arbitrary policy π
2: while π ≠ π′ do
3:    Compute the Value for the policy π
4:    for all s along policy π do
5:       V_π′(s) = J(s, π) + V_π(s′)
6:    if V_π′(s) < V_π(s) then
7:       π = π′
```

Algorithm 1.2 works by first choosing an arbitrary policy from the current state to the goal [3, 4]. Then working backwards from the goal along the current policy searching for better policies. In the search try different actions and calculate a value for the new policy created by that action. If the new policy performs better than the current best policy update the policy.

Ricard Torres published a MATLAB script implementing both policy- and value- iteration [4]. Torres' work provides insight to how to programmatically find the optimal policies once the network is known but the organization of the network is done in a non programmatic manner. The approach devised and presented in this thesis builds on his work. First by developing a way to programmatically organize the Dynamic Programming problem in to a cost network. Second by organizing the connections between the nodes of network into a hierarchically organized network of connections, a connection network, to reduce computational time.

1.2 The Rowdy Runner

The Rowdy Runner is a torso actuated rimless wheel robot. Originally developed by the undergraduate senior design team FOA, the Rowdy Runner's purpose is to facilitate research on rimless

wheels as a method of legged locomotion. FOA's Rowdy Runner consisted of 2 rimless wheels locked in rotation by a single axle. The torso of Rowdy was mounted on the axle by freely rotating bearings. Coupled to the axle via a toothed output sprocket through an untensioned belt was a single motor mounted in the torso. This version of Rowdy was able to produce motion by controling its body angle. The current iteration of the Rowdy Runner is able to produce these motions at a higher resolution and with greater reliability.

Consider the case where the Rowdy Runner is a black box where the observe or actor sees only the input and response. This thesis presents a Dynamic Programming approach to determine optimal control policies that provide inputs to the black box, given some response, to produce a desired response. The test case is a model of Rowdy where the wheels are locked in rotation and Rowdy is capable of controlling the angle of its torso, allowing motion in only one plane. While this level of control is sufficient to produce an indirect method of controlling its speed, the objective is to directly control speed. This research targets a 2D planar model of the Rowdy Runner II, an iteration on FOA's contribution, to investigate the application of Dynamic Programming as a method of speed control and further, as a standard to which the performance other control methods may be evaluated.The current Rowdy Runner was developed by Eric Sanchez. It is an independently torso-actuated rimless wheel robot. It is the second iteration of its design. It consist of two rimless wheels, each actuated by a motor housed in the torso.

CHAPTER 2: LITERATURE REVIEW

Dynamic Programming is a complex set of mathematical tools used to find optimal paths between the states of a discrete system. First described by Richard Bellman, Dynamic Programming uses a "divide and conquer" approach to separate complex problems to multiple simpler problems which represent states of the complex problem [6]. The solution to these simpler problems is the optimal policy to the larger problem. It is the found via an understanding that

"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision" [7, 8].

This is referred to the Principle of Optimality [7]. Commonly, these decision making problems with which Dynamic Programming deals are organized into a network of states whose decisions transition them to other states. The optimal paths through these network are found using Markovian Decision Process to recursively solve Bellman Equations [1, 3, 6].

The discretization method used in this thesis is similar to the one described in Chis Atkeson's paper [9]. Here he and others applied Dynamic Programming to solve the problem of velocity control for a 2-D bipedal walker with a torso. They circumvented the "Curse of Cimensionality" by reducing the number of states, considering only an instance of the robots periodic state orbit. With this reduced state model, he used the velocity of the robot at the mid-stance as the target variable in optimization. An interesting concept presented in this paper is the composition of the the cost function, where input torque, mid-stance velocity, and, more interestingly, the acceleration of the swinging leg as a function of step time are considered. When applied this method as stated "can handle starting from zero velocity, and can achieve a range of velocities." The performance of the control policies between these velocities depends on the optimization criteria expressed in the cost function and the resolution of the grid. One of the shortcomings mentioned about this approach, was that certain dynamical behaviors had to be overlooked in the reduction of state. Examples include the contribution of the torso to the system's dynamical behavior and the ankle

push off in each step. In conclusion Atkeson states that complex nonlinear control laws can be produced in solutions to Dynamic Programming problems.

This reduction of state approach can be seen again in Pranav Bhounsule and Ezra Ameperosa 2016 ASME paper [10] where they used a dead-beat controller to target a poincare map for quicker response in correcting disturbances in the gaits of rimless wheels. By switching among the limit cycles, the walker was able to quickly change its speed. This work also takes inspiration from their approach to the modeling and simulation of a rimless wheel robot. In their paper they break up the rimless wheel robot's equations of motion in to two phases: single stance and heel strike. The problem with dead-beat control is that it is very sensitive to modeling errors.

Bhounsule and Zamani proposed an exponential orbital stabilization controller that reduces the modeling errors exponentially over a few steps rather than in a single step [11, 12]. Their exponential orbital stabilization controller is robust to the modeling errors unlike the dead-beat controller.

One way to change speed of a walker is to create multiple limit cycles and switch among them as in the work by Bhounsule et al [13]. In their work, they create a sufficient number of limit cycles corresponding to different velocities between the start speed and the goal speed.

Tad McGeer conducted research on passive dynamic walking machines that had groundbreaking results [14]. McGeer found and described a class of bipedal machines that have natural dynamic modes of walking and capable of passively walking using only gravitational potential energy provided by the environment. He began his research analyzing and studying rimless wheels. He looked at the behavior of a rimless wheel throughout its gait and described its motion before and after collision using a now ubiquitous approach where he assumes that angular momentum is conserved by considering the ground-foot collisions are inelastic impulses. The equations he derived described the dissipative nature of the rimless wheel's steps. These equations are modified for use in this work. McGeer pioneered the field of dynamic walking with his paper [14]. This thesis builds on McGeer's work by adding an actuated torso to rimless wheels exploiting its natural dynamic mode to produce dynamic walking on level ground and change speeds with some measure

of agility.

CHAPTER 3: APPROACH

This chapter of this document covers the 3-step methodology devised to organize a Dynamic Programming problem and find globally optimal policies. The development of this methodolgy was grossly motivated by the use of MATLAB's parallel computing toolbox and MATLAB's profficiency in handling 2D matrices. Also discussed in this chapter are the insights gained from the application of this methodology to two different systems along with the short comings of Dynamic Programming and methods used to circumvent these faults.

3.1 Generating the Cost Network

Step 1 of the 3-step methodology is generating the cost network. The cost network is an organized network of interconnected nodes that represent states of a model, hold information about the states and available transitions from the states.

3.1.1 Nodes of a Network

Algorithm 3.1 Class nNode

DATA MEMBERS

```
1: ID = []
2: state = []
3: connections = {[]}
4: optimal_value = []
5: optimal_policy = []
```

Nodes of a network are discrete containers composed of 5 data members: a *node ID*, a *state*, an *optimal policy*, an *optimal value*, and *connections*. The *node ID* data member is a value or an array of values are used to reference the node's position within the network. This *node ID* correlates to the state data member. The *state* is a unique combination of discrete and indevisible state variable values used to describe the model's configuration. The *optimal policy* is a data member which stores the ID of the node that correlates to the subsequent state in the globally optimal policy. The *optimal value* data member is a container which stores the total cost to go from the current node

to the goal node. The values of the *optimal policy* and *optimal value* data members are set and updated in the evaluation of the cost and connection network step. The *connection* data member is a container that stores information about connections to subsequent nodes/states in the form: { [*state ID*] (*action*) (*cost*)}, where *cost* is the cost accrued during the transition from the current state to the state stored in the node referenced by *state ID* when applying the action stored in *action*.

3.1.2 Cost Network Generation

The first step in generating the cost network is developing a simple yet sufficient model of the system that captures its interesting behaviors. For physical systems like the Rowdy Runner we are most interested in their dynamics. Generally, to create a model begin by observing the system in a coordinate frame, determine how many variables are necessary to describe its unique state, and define equations that describe the evolution of its state through time i.e. the equations of motion of the system. In order to reduce modeling errors, a simple approach is to compare the behaviors of the model and the system and define and constrain the state space of the model to a region in which the equations of motion and the behavior of the physical system agree with minimal error. This region is referred to as the region of feasibility. From here there are three choice that produce grossly different results. One, discretize the feasible space using the state variables and allow time as a variable. Two discretize the equations of motion using time allowing variable time between states. The third choice which could possible overconstrain the problem is to discretize both time and state truncating differences. Each of these approaches will produce a finite number of states in a bounded space. Also by taking only the action that produces the lowest cost action necessary to transition between states that is capable of being produced by the system the number of actions i.e. connections to later evaluate is reduced. After discretizing the feasible space and the actions the cost network is generated using a process described by Algorithm 3.4. Generate a blank $N_1 \times N_2 \times \cdots \times N_{(n-1)} \times N_n$ network of nodes for n state variables where N_i is the number of nodes that correlate to S_i representing every discrete increment in the feasible space along that state variable. Evaluate each unique state-action combination using the transition function and

Algorithm 3.2 Compare Connections

INPUT (connection c, Node **n**)

for all conn $\in$ **N**([s, **s**]).connections **do**
  i = i + 1
  if conn[1] == connection[1] **then**
    if conn[3] > connection[3] **then**
      The connection to the state in c is of lower cost. Update it.
      return i
  else
    The connection to the state in c is of higher cost. Leave the currently stored connection.
    return NULL
The connection to the state in c does not exist so add it.
return i

Algorithm 3.3 costEval

INPUT (state s, states **S**, actions A, network **N**)

dims = *size*(**S**)
if dims[2] > 1 **then**
  for all s $\in$ **S**(:,1) **do**
    costEval([s, **s**], **S**(:,2:end), A)
else if dims[2] = 1 **then**
  for all s $\in$ **S do**
    for all a $\in A$ **do**
      i = 1
      (s', J) = **T**([s, **s**], a)
      connection = $\{s', a, J\}$
      for all conn $\in$ **N**([s, **s**]).connections **do**
        i = i + 1
        if conn[1] == connection[1] **then**
          if conn[3] > connection[3] **then**
            N([s, **s**]).connections[i] = connection
        else
          continue
      N([s, **s**]).connections[i] = connection
else
  break

Algorithm 3.4 Generate Cost Network

INPUT(states **S**, *path/to/file* to store the Cost Network)

network = $N_1 \times N_2 \times \cdots \times N_{(n-1)} \times N_n$ for n = # of state variables
for all s $\in$ **S**(:,1) **do**
  costEval([s, **s**], **S**(:,2:end), A, **N**)

calculate the cost accrued by the transition to state s'. Then populate the data in the node storing the lowest cost unique connection for every state-action-state combination $\{s, a, s'\}$.

3.2 Generating the Connection Network

Step 2 of the 3-step methodolgy is the generation of the connection network. The Connection Network is a hierarchically organized network of connections between nodes arranged in stages of immediately connected nodes. At the head of the network or the 0^{th} stage is the goal node. The next stage is home to the connections linking the nodes immediately connected to the goal node, stored in the form $\{s'_i, a_j, s_{goal}\}$. The second stage is composed of the connections of nodes with immediate connections to nodes with connections in the first stage, $\{s''_i, a_j, s'_k\}$. Repeatedly following this line of succession the tail stage is composed of the connections of nodes furthest disconnected from the goal node at the head, $\{s^n_i, a_j, s^{n-1}_k\}$. The Algorithms 3.5 and 3.6 outline the process of programmatically building a connection network. Archetypically, the connection network is generated by repeatedly searching the cost network for nodes with connections to nodes who have connection stored in the previous stage. If the connections found are not already recorded in the cost network store them in the current stage. This step in the process of Dynamic Programming is motivated in an effort to reduce computational time by parallelizing the process of evaluating networks. It was observed that connections on the same stage are independent from one another and may be evaluate simultaneously. This idea proved fruitful.

Algorithm 3.5 Network search and retrieve

INPUT State: s^*, Cost Network: S
OUTPUT Connection Network Stage C

```
for all s ∈ S do
  for all a ∈ A do
    if T(s, a) = s* and {s, a} ∉ C then
      C.append({s, a})
```

Algorithm 3.6 Connection network generation

INPUT Cost Network: S
OUTPUT Connection Network $\mathbf{C}$

1: $i = 1$
2: **Initialize goal state value function:** $V(s^*) = 0$
3: $\mathbf{C}[i] = NetworkSearchAndRetrieve(s^*, S)$
4: $i = i + 1$
5: $N_{cos} = numberOfConnectionsIn(S)$
6: $N_{con} = length(C[i])$
7: **while** $N_{con} < N_{cos}$ **do**
8: **for all** $c \in \mathbf{C}[i-1]$ **do**
9: $C.append(NetworkSearchAndRetrieve(s^*, S))$
10: $\mathbf{C}[i] = C$
11: $N_{con} = N_{con} + length(C)$

3.3 Evaluating the Connections

The final step in this process is to evaluate the cost and connection networks in tandem to find an optimal policy. This method is considered to use a combination of policy- and value- iteration to find the optimal policy.

Instead of recursively generating value functions that are functions of other value functions like in value iteration. Connecion evaluation instead looks only for nodes that have an optimal value set. This is possible as a result of the heirarchical structure of the connection network. Since each stage of the connection network is composed of the connections that are immediately connected to the next previous stage, the connections contained in each stage are independent of one another. Therefore, they may be evaluated simultaneously. This also means that when following the heirarchy there will be no need for generating nested value functions and the required active memory will be reduced.

Algorithm 3.7 Evaluate the cost and connection network

INPUT Connection: $\{s, a, s'\}$, Cost Network: $\mathbf{S}$
OUTPUT connection: c^*

1: **if** $V(s, a) < V(s, a^*)$ **then**
2: $c^* = \{s, a, s'\}$
3: **else**
4: $c^* = NULL$

Algorithm 3.8 Evaluate the cost and connection network

INPUT Cost Network: $\mathbf{S}$, Connection Network: $\mathbf{C}$
OUTPUT Updated Cost Network: $\mathbf{S}^*$

1: **for all** $n \in \mathbf{C}$ **do**
2: i=1
3: **for all** $m \in \mathbf{C}[n]$ **do**
4: $C[i] =$ *evaluateConnection*$(\mathbf{C}[n][m], S)$
5: $i = i + 1$
6: **for all** $c \in C$ **do**
7: **if** $c \neq NULL$ **then**
8: set the action at state $c[1]$ equal to $c[2]$

The process goes, for each connection in the connection network, compare the value of that connection with the value of the current connection stored. If the new connection produces a better value, update the optimal_policy and optimal_value stored at that node in the network otherwise skip it.

CHAPTER 4: APPLCATION OF METHODS

4.1 Pendulum Upswing

The goal in this exhibition of Dynamic Programming is to determine a globally optimal control policy to perform an upswing from 0° to 180° on a simple torque-controlled pendulum. The purpose for this digression is to better visualize and understand Dynamic Programming's process and results. The actuated simple pendulum was chosen because "*the dynamics of a single pendulum are rich enough to introduce most of the concepts from nonlinear dynamics*" and the pendulum relatability to our system [15, 16].

Beyond verifying that the problem has a substructure capable of being optimized Dynamic Programming requires 4 components; *S* a set of discrete states that describe the configuration of the system, *A* a finite set of actions the system is capable of making, a transition function $\mathbf{T}(\mathbf{s}, \mathbf{a})$ that when given a state-action pair returns the next state, and a Cost function that penalizes the system for undesired behaviors [9, 16].

4.1.1 Generating the Cost Network for a Simple Pendulum

Here we generate the equations of motion that give a mathematical description the behavior of the system. As expressed in Figure 4.1 the pendulum consist of a mass *M* at then end of a long massless rod of length *l* attached to a frictionless pin joint. The joint's positive direction of rotation is along the y-axis $\hat{i} \times \hat{z}$ restricting the motion of the pendulum to the x-z-plane. A 0° angle is set where the pendulum points along the positive z-axis. An actuator is placed at the pin joint has a bidirectional maximum output τ_{max}. The pin joint is a frictionless.

With the model as described in Figure 4.1 Lagrangian mechanics was found to be best suited for deriving the equations of motion rather than Newtonian for two reasons. First, all the forces described in the model are conservative. Second, while Lagrangian and Newtonian approaches are both capable handling conservative systems, due to the scalar nature of the Lagrangian approach it isn't so cumbersome when working with the polar coordinate systems other than cartesian [17,

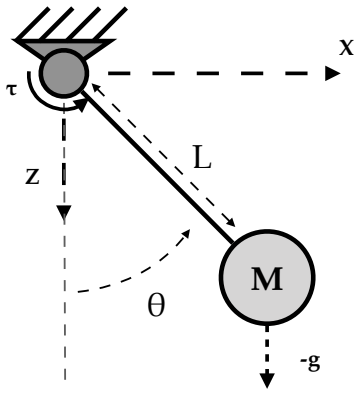

Figure 4.1: Model of simple pendulum

18]. The polar coordinate system was chosen because with polar coordinates the pendulum's state can be described by two variables $\left(\theta, \dot{\theta}\right)$ rather than the four $(x, \dot{x}, y, \dot{y})$ required by cartesian coordinate system.

Pendulum Equations of Motion

The Lagrangian is equal to the difference between the kinetic and potential energy of the pendulum as shown in Equations 4.1. The potential energy of the system is a function of the pendulum's angle θ defined as the product of the mass, the gravitational acceleration constant, and the cosine of the pendulum's angle with respect to the z-axis as described in Equations 4.2. The kinetic energy of the system is also a function only of θ defined as half of the pendulum's inertia times the pendulum's squared rotational velocity as described in Equations 4.3.

$$L = T - V \tag{4.1}$$

$$V = m\, g\, l\, \sin\left(\theta\right) \tag{4.2}$$

$$T = m\frac{\left(\dot{\theta} L\right)^2}{2} \tag{4.3}$$

$$L = m\frac{\left(\dot{\theta}L\right)^2}{2} - m\,g\,l\,\sin\left(\theta\right) \tag{4.4}$$

The next steps in derivation of the equations of motion are to use Equations 4.5 to derive the forces acting within the system. The result of this step is shown in Equations4.6.

$$\frac{d}{dt}\left(\frac{dL}{d\dot{\theta}}\right) - \frac{dL}{d\theta} = 0 \tag{4.5}$$

$$m\,L^2\,\ddot{\theta}\left(t\right) + m\,g\,L\,\sin(\theta) = 0 \tag{4.6}$$

The torque output by the actuator Next add the forces acting within the system to the actuator for acting on the system to produce the finalized equations of motion that will be used as the transition function in Dynamic Programming to form Equations 4.7.

$$m\,L^2\ddot{\theta}(t) + m\,g\,L\,\sin(\theta(t)) = \tau_{act}(t) \times \vec{r} \tag{4.7}$$

$$\mathbf{x}_{k+1} = \int_t^{t+\Delta t} \begin{bmatrix} \dot{\theta}\left(t\right) \\ -\frac{g}{l}sin\left(\theta\left(t\right)\right) - \frac{\tau}{l} \end{bmatrix} dt \tag{4.8}$$

Afterwards we discretize the equations of motion using time by integrating the equations of motion over the increment of time Δt to produce the time-discretized equations of motion shown in Equations 4.8.

Pendulum State Space

Now with a function that describes the behavior and evolution of the system over time we must define boundaries to eliminate unnecessary computation [9, 16]. We will refer to the region within the boundaries as the region of feasibility [19]. This term describes a state space where all included states the satisfy all the imposed constraints. Outside this region, for real systems, the system may fail to behave as described by the equations of motion and produce modeling error. There are two state variables that make up this region the angular position and the rate of change of the angular

Pendulum Type	**Angle** $\theta\ (rad)$			**Angular velocity** $\dot{\theta}\ (\frac{rad}{s})$			**Actuator** $(N\,m)$			$\Delta t(s)$
	min	max	grid	min	max	grid	min	max	grid	
Strong	0	2π	150	-6	6	150	-10	10	200	0.1
Weak	0	2π	300	-10	10	300	-1.5	1.5	30	0.05

Table 4.1: Pendulum parameters

position. Positionally, the pendulum has the ability to move freely between any 0 and 2 π radians. There are no "real" positional limitations i.e. the feasible space is bound but infinite. This is because in a real circle all positions between and including 0 and $2\,\pi$ are equivalent all positions between and including $2\,n\,\pi$ and $2\,\pi\,(n+1)$ where n is any integer value. Numerically, angular positions which lie outside these bounds are wrapped to the range $\{0, 2\,\pi\}$. The pendulum's angular velocity state variable $\dot{\theta}$'s feasible region is between angular velocities ranging from -10 to 10 $\frac{rad}{s}$, about 95 rpm.

Using the equations of motion we use time to discretize the Equations of motion and produce a transition function, $T\,(s, a)$, that when given a state and an action produces a new state. With this transition function and the discretized state space we can now build our cost network.

Two classifications of pendulum are considered for optimization: sufficiently actuated and weakly actuated. The sufficiently actuated will not require a complex nonlinear control law to reach the goal. The sufficiently actuated pendulum has a maximum output torque capable of holding the 1 kg mass at $90°$. The weakly actuated pendulum is only able hold the 1 kg mass about $10°$ in respect to the gravity vector.

4.1.2 Generating the Connection Network for a Simple Pendulum

Following the process described in section 3.1 the pendulums cost network generated using the values recorded in Table 4.1. Figures 4.2 provide a visual representation of the blank cost network. Here the z axis of the cylinder represents the pendulums velocity and the angle away from 0 cylinder

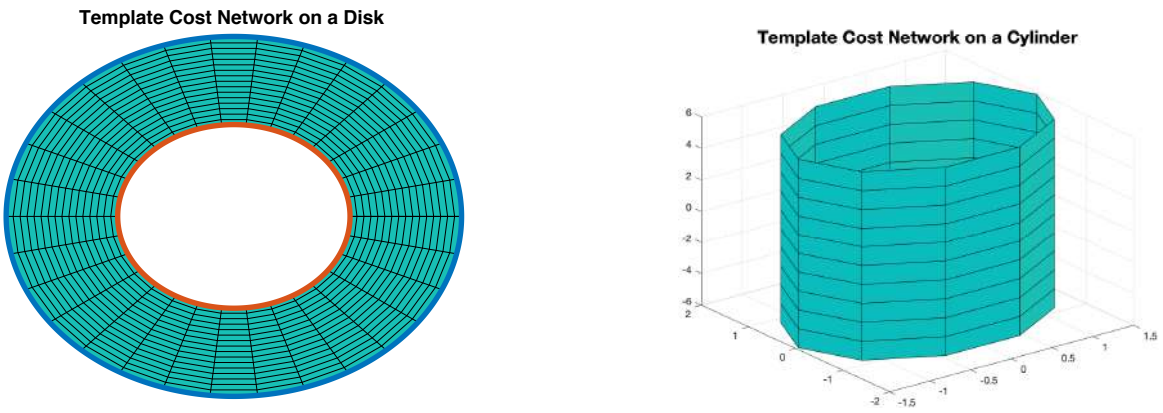

Figure 4.2: Low resolution pendulum network template represented on a cylinder and disk

4.1.3 Evaluating the Connection Network for a Simple Pendulum

To find the optimal paths from all of the states in the network to the goal, evaluate the cost and connection networks using the process previously described. Woring in a top down manner visiting each stage of the connection network and comparing the connection stored there with the currently stored value. In the case that this connection produces a policy with a lower cost to go to the goal we update the network. For the pendulum there are a large number of connections to evaluate especially in the case of the sufficiently actuated pendulum which has 200 possible actions per state. Due of the higher number of decisions it is expected that it will require more iterations evaluating the cost and connection netowrks to reach the stopping criteria expect this to require more cost and connection network evaluation iterations to wholly refine a globally optimal policy.

4.2 Rowdy the Rimless Wheel Robot

The Rowdy Runner is a torso actuated rimless wheel robot. Originally developed by the undergraduate senior design team FOA the Rowdy Runner's purpose is to facilitate research on rimless wheels as a method of legged locomotion. The Rowdy Runner is only capable of controlling its body angle to produce which is an indirect method of controlling its speed. This research references an iteration on FOA's contribution, The Rowdy Runner II, to investigate the application of Dynamic Programming as a method of speed control and further, a standard to which the perfor-

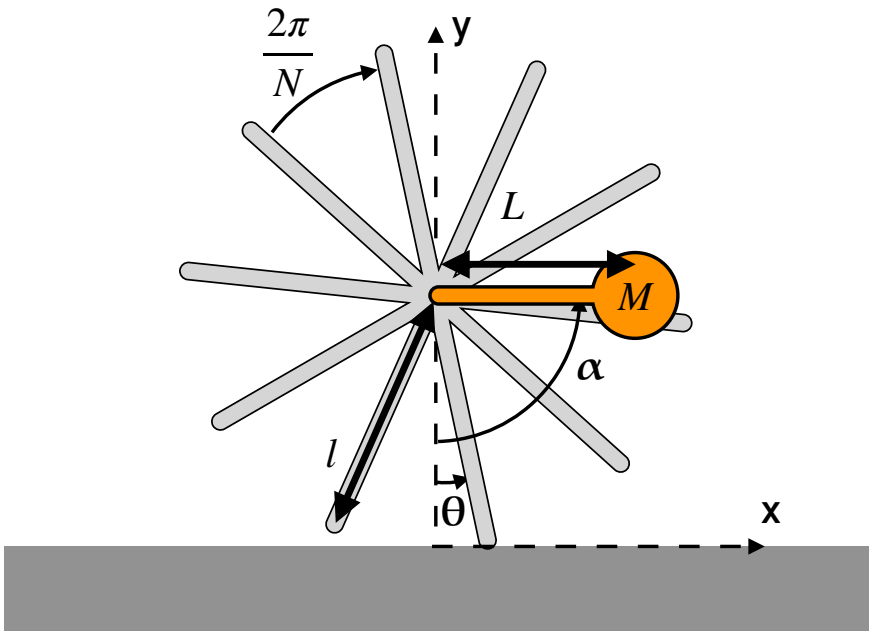

Figure 4.3: Model of torso actuated rimless wheel robot

mance other control methods may be evaluated.

4.2.1 Model of Rowdy

The Rowdy Runner is a torso actuated rimless wheel robot. Originally developed by the undergraduate senior design team FOA the Rowdy Runner's purpose is to "facilitate research on rimless wheels as a method of legged locomotion" [20]. The Rowdy Runner is capable of controlling its body angle to produce motion but currently there is no direct method for speed control. This research references an iteration on FOA's contribution, The Rowdy Runner II, developed by Sebastian Sanchez and UTSA's RAMLab to investigate the application of Dynamic Programming as a method of speed control and further, a standard to which the performance other control methods may be evaluated. The rimless wheel robot was modeled as shown in Figure 4.3. The rimless wheel is a discrete wheel with N = 10 massless spokes each of length l and a mass of m centered at the hub. The torso is modeled as a point mass held at length L away from the axis through the wheel hubs.

4.2.2 Equations of Motion

The dynamical behavior of Rowdy can be described in 2 phases; the stance phase, and the heel-strike phase. The stance phase is characterized by Rowdy having only one foot in contact with the ground.

Stance Phase

During this period Rowdy pivots about the foot of the stance leg in the forward direction. This motion is analogous to the behavior of an inverted double pendulum. The equations of motion of the system during the stance phase were derived by balancing its angular momentum about the contact foot. The results are presented in Equations 4.9,4.10,4.11.

$$\mathbf{A}\ddot{\theta} = \mathbf{b}_{ss} \tag{4.9}$$

$$\mathbf{A} = M\,l^2 + l^2\,m - L\,M\,l\,\cos\left(\alpha - \theta\right) \tag{4.10}$$

$$\mathbf{b}_{ss} = \left(M\,g\,l + g\,l\,m\right)\,\sin\left(\theta\right) + L\,M\,l\,\dot{\theta}^2\,\sin\left(\alpha - \theta\right) - L\,M\,g\,\sin\left(\alpha\right) \tag{4.11}$$

Heel Strike

The heel-strike phase begins when the leading foot collides with the ground. During this phase a change of support occurs and the leading foot becomes the stance foot. The collision that initiate this phase is considered to be in elastic and impulsive [9,13,14,21]. By performing an angular momentum balance about the collision location, the angle leg angle after heel-strike can be described by Equation 4.12 and the angular rate after heel-strike can be described by {4.13, 4.14, 4.15} .

$$\theta_{after} = -\theta_{before} \tag{4.12}$$

$$\mathbf{A}\,\dot{\theta}_{after} = \left(\,(M+m)\;l^2\mathbf{\Theta} + M\,L\,l\,\mathbf{\Phi}\right)\dot{\theta}_{before} \tag{4.13}$$

$$\mathbf{\Phi} = \cos(\alpha)\;\cos(\theta) + \sin(\alpha)\;\sin(\theta) \tag{4.14}$$

$$\mathbf{\Theta} = \sin(\theta)\;\sin\left(\theta + \frac{2\pi}{N}\right) + \cos(\theta)\;\cos\left(\theta + \frac{2\pi}{N}\right) \tag{4.15}$$

Phase Transition Criteria

This phase transition from single stance to heel strike occurs when the condition expressed in Equation 4.16 is met. The poincare section of the limit cycle of the system is taken at the mid-stance when ϑ = 0 as described by Equation 4.17.

$$l\cos(\theta) - l\cos\left(\theta + \frac{2\,\pi}{N}\right) = 0 \tag{4.16}$$

$$\theta = 0 \tag{4.17}$$

Failure Criteria

There are four failure criteria which indicate a failed state transition or the transition to an impossible state or a state that immediately leads to one; *Falling Back*, *Ground Penetration*, and *Flight* [10]. Only the forward motions of Rowdy are expressed in the model so rolling backwards leads to a failed state. *Falling Backwards* is described by Equation 4.18. No part of model should pass through the ground plane. The condition of *Ground Penetration* described in Equation 4.20 detects this and returns a failure flag it is met. A *Flight Condition* occurs when the ground reaction forces in the y-direction are less than or equal to 0. Should this occurs Equation 4.19 flags the failure state. During simulation each of the failure criteria are checked and prompt the observer that they have occurred, not stopping integration. This is to gather information about the model and equations of motion via visual representations.

$$\theta > \frac{\pi}{N} \tag{4.18}$$

$$F_Y = (M+m)\ \left(-l\ \cos\left(\mathrm{gam} - q\right)\ u^2 + g + l\,\mathrm{ud}\ \sin\left(\mathrm{gam} - q\right)\right) \tag{4.19}$$

$$\theta < \frac{\pi}{N} \tag{4.20}$$

Step to Step Discretization

In this approach we reduce the number of state variables required to define the system in a method similar to the one described in Chris Atkeson's paper [**?**]. We simplify the system state by targeting Rowdy's poincare map, effectively defining an interdependency between the current step and the next step via the periodic orbit of the system's state. The poincare map of the system is defined using Equationss 4.9 - 4.17.

Simulation

Under these conditions we see the velocity asymptotically approach a constant value for a constantly held torso angle, characteristic of the behavior of rimless wheels. This indicates an accurate modeling of the dissipative collisions of steps.

4.2.3 Organizing the Dynamic Programming Problem for Rowdy's Speed Control

Here we use the methods described earlier in this section to organize the Dynamic Programming problem by first generating the cost and connection networks and subsequently evaluating those network to find optimal paths. This planar torso-actuated rimless wheel robot is more complex than the simple pendulum, with 2 degrees of freedom the total number of state variables 4. This higher dimensionality would likely require much more computational time using the methods described This process with its reduced state provided a the above describe equations of motion to for a transition function that relates the mid-stance velocity at one step to that of the next.

Speed of leg at mid-stance $(\frac{rad}{s})$			**Input torso angle** $(^\circ)$		
min	max	grid	min	max	grid
-4.9312	-0.0406	150	0	90	90

Table 4.2: Rimless wheel robot parameters

Generating Rowdy's Cost Network

When trying to find a state space the computational time used while trying to find a good state space was reduced by discretizing the feasible space and actions using the lowest possible resolution. We chose a grid size of 1 degree increments between 0 and 90 degrees for the torso angle. After our work with the pendulum we developed a method to find boundaries and discretize a state space that would provide good network health. This term of network health will be discussed in detail in the next subsection 4.2.3.

We arbitrarily chose large positive and negative rotational velocities, discretized the space at a low resolution. Using the step-to-step equations of motion we evaluate the states with each control action to determine if a state transition is possible (no failure criteria was triggered). We moved the boundary inward maintaining the same grid size if possible transitions weren't found at the edges and expanded the boundary and increasing grid size if possible states were found. After pruning states that were not reachable and did not connections to other state we found that a grid size of 150 for a boundary between 0.67 rad/s and -5.6 rad/s was a proper definition of the discretized state space for this model.

The cost network for Rowdy is generated by discretizing the state space using the parameters from Table 4.2

Generating Rowdy's Connection Network

Rowdy If we used the same grid size as the in the pendulum this would produce N^4 unique states each with a set of actions *A* that need to be evaluated. For a pendulum, where the number of unique states for a uniform grid of count *N* is N^2 with N = 150 and 200 possible actions required 2 hrs and 27 mins to populate the network with optimal policies.

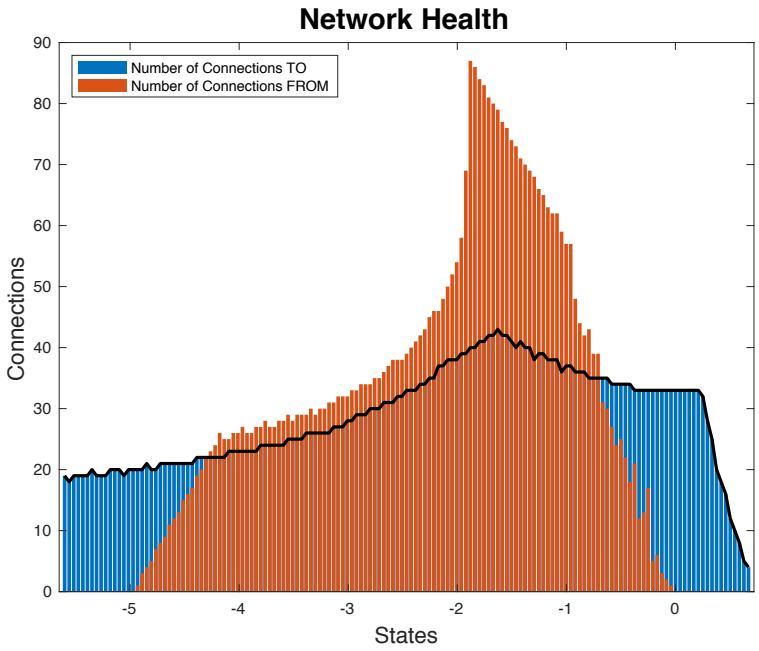

Figure 4.4: Network interconnectivity

leaves 4.5\,\times\,10^6 possible combination. Therefore some adjustment to out method is necessary. Ideally we would like to reduce the number of state variables yet still be able to successfully control the system.

Figure 4.4 depicts the interconnectivity of the rimless wheel robot's cost network. Since network evaluations and the refinement of policies will not affect the number of connections observing the number of connections between states and with which states they occur may be a good measure of a networks health.This network health is believed to be related to the quality of a network's optimal substructure. An η too large and the network is over constrained and too low may be over constrained. The each increment on the x axis represents some state s and the value on the y axis is the number of other states S' that have an immediate connection to state s.

This was found during generation of the connection network, by searching the network as described in the Algorithm 3.5 we counted the number of states with actions that transitioned to each state. We found that for the states that had fewer connections it was more likely that limit cycles that allowed them to maintain their velocity between poincare sections would not be found.

This is because there is no available action (torso angle applied over a complete step) that allows that step to begin and end at the same velocities. These states with fewer connections lie in regions bordering the unreachable boundary. Outside of this region there exist only states that are in fact feasible but cannot be reached i.e. there are no states in the network that have actions capable of reaching them. Therefore a healthy network would have a high number of connections though too many connections produces the issue we're attempting to avoid in targeting the poincaré section for control. Too many connections would require more computational time since each connection must be evaluated to conclusively find a globally optimal policy.

Evaluating Rowdy's Connection Network

Evaluation of Rowdy's cost network required a single iteration, one initial iteration followed by a second that verified that the optimal path had been found, in the case of optimal control policy that transitioned from $-2\frac{rad}{s}$ to $-3\frac{rad}{s}$. This may be an indication of an over constrained situation, that too few policies between transitions exist. Dynamic Programming gives better performing solutions where there are more competing policies.

CHAPTER 5: RESULTS & DISCUSSION

The three step methodology discussed in the previous section was used to organize and solve a Dynamic Programming problem for two system to minimize a weighted sum of the square of the deviation from the desired state and the actuator effort. The first system is a simple pendulum where the goal is to transition from $\{0, 0\}$ to $\{\pi, 0\}$. Two variants of this system were created by imposing different actuator constraints in order to visualize behavior of the cost network as parameters are changed. The second, is the case of a planar torso-actuated rimless wheel robot whose dynamics require more state variables to describe. The results of these trials in Dynamic Programming show that for a full state pendulum we were able to determine the optimal policies for the upswing of both a strongly and weakly actuated pendulum were able to be determined. One issue, however, is that the accuracy of Dynamic Programming solution depends on the fineness of the grid size. Though not insurmountably problematic for the simple pendulum the planar torso actuated rimless wheel robot, with its 2 more degrees of freedom at grid as fine as the pendulum's would require a prohibitively large amount of time to organize and solve. To circumvent this issue, we use a practice, common in the area of legged robotics, reducing the state by targeting the solution to a poincaré section [9, 10].

5.1 Pendulum Results

In two separate cases of a simple actuated pendulum Dynamic Programming was used to determine globally optimal control policies to perform upswings from 0° at 0 $\frac{rad}{s}$ to 180° at 0 $\frac{rad}{s}$. Each of the pendulum's actuators were designed to hold their mass steady at some angle in the first quadrant when applying their respective maximum torques. The strong pendulum is sufficiently strong such that it can hold its mass at any angle. In the case of the weak pendulum however, actuator limitations are imposed such that the pendulum is only capable of holding steady at an angle no greater than 10°. The considerations in the discretization of the state space from which the cost and connection network were built are as stated. The velocity and speed resolutions should be

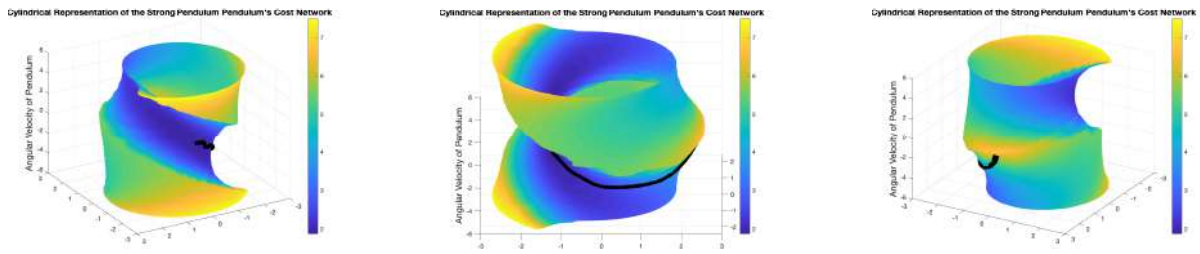

Figure 5.1: Visualization of sufficiently actuated pendulum's cost network on discoidal and cylindrical surfaces with optimal policy over laid in black

sufficiently small such that the smallest action integrated over the time step should produce a measurable change. Here presents the dilemma, the higher the resolution the better the performance but the greater the amount of time required to perform Dynamic Programming.

The results for the Dynamic Programming solution of the pendulum show the capabilities of Dynamic Programming to find complex nonlinear solutions to difficult problems but also shine a light on its the shortcomings.

5.1.1 Strong Pendulum

The resolutions of the strong pendulum are recorded in Table 4.1. Shown in the figures are the cylindrical and discoidal representations of the cost network. The cylindrical representation of the cost network 5.1 is an expression where: the the z-axis indicates rotational velocity, the angle made by the ratio of the x and y axis, $\theta = \arctan(\frac{x}{y})$, represents the angular position of the pendulum, and radius $r = \sqrt{x^2 + y^2}$ represents the cost accrued by traveling along the optimal policy from that state to the goal. The discoidal representation of the cost network where: the angle made by the ratio of the x and y axis represents the angular position of the pendulum, the radius correlates to the velocity where the innermost represents the lowest velocity and the outermost represents the

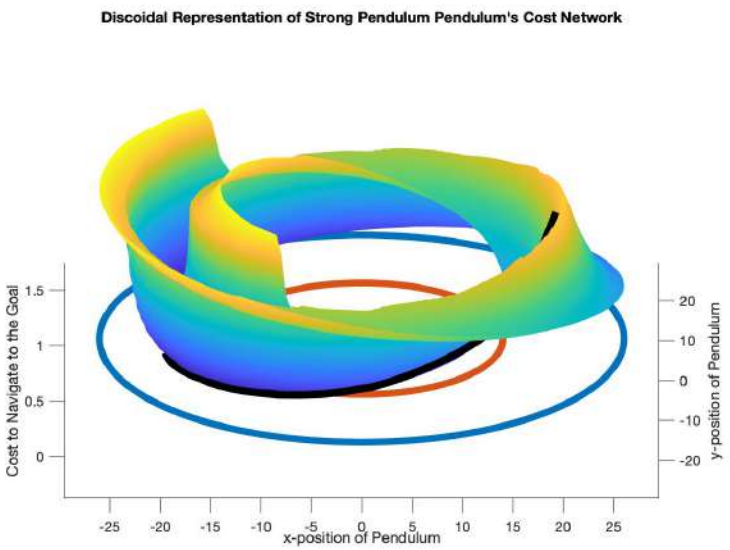

Figure 5.2: Visualization of weakly actuated pendulum's cost network on discoidal surface with optimal policy over laid in black

Phase	Time Required
Generating Cost Network	0h 16m 51s
Generating Connection Network	1h 38m 31s
Evaluation of Connections	31m 49s
Total Time Required	2h 27m 11s

Table 5.1: Computational time to organize and solve strong pendulum Dynamic Programming problem

max velocity. The computational time required to organize and solve the Dynamic Programming problem is recorded above. Generating the cost network includes evaluating all the 150×150 states for all 200 actions. There are fewer connections in the connection network because there are a number of different actions which transition to the same state.

5.1.2 Weak Pendulum

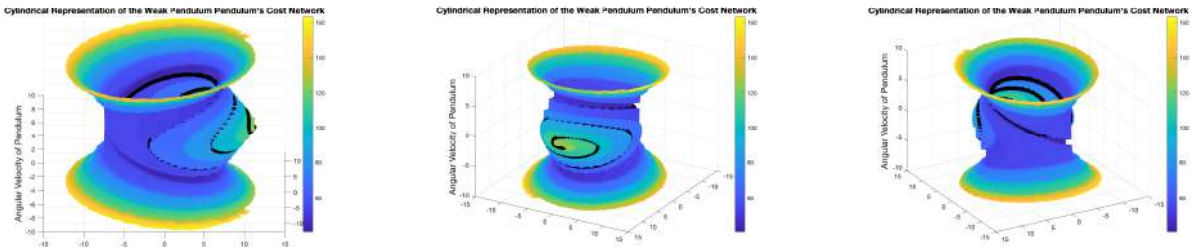

Figure 5.3: Visualization of weakly actuated pendulum's cost network on cylindrical surface with optimal policy over laid in black

The resolutions of the weakly actuated pendulum are recorded in Table 4.1. Shown in the figures are the cylindrical and discoidal representations of the cost network. The cylindrical representation of the cost network 5.3 as in the representations of the strong pendulum's visualization are expressed as: the the z-axis indicates rotational velocity, the angle made by the ratio of the x and y axis, $\theta = \arctan(\frac{x}{y})$, represents the angular position of the pendulum, and radius $r = \sqrt{x^2 + y^2}$ represents the cost accrued by traveling along the optimal policy from that state to the goal. The discoidal representation of the cost network where: the angle made by the ratio of the x and y axis represents the angular position of the pendulum, the radius correlates to the velocity where the innermost represents the lowest velocity and the outermost represents the max velocity.

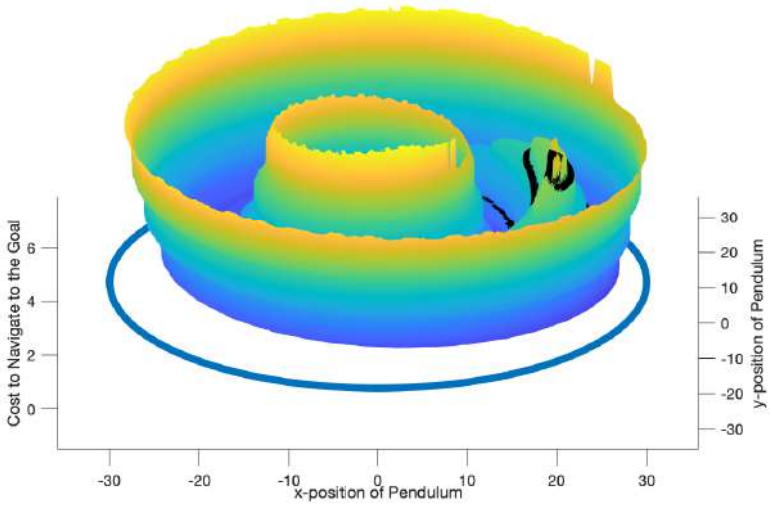

Figure 5.4: Visualization of weakly actuated pendulum's cost network on discoidal surface with optimal policy over laid in black

Phase	Time Required
Generating Cost Network	0h 10m 49s
Generating Connection Network	3h 30m 17s
Evaluation of Connections	50m 19s
Total Time Required	4h 31m 25s

Table 5.2: Computational time to organize and solve weak pendulum Dynamic Programming problem

5.2 Speed Control of the Rowdy Runner II

This approach produces a step-to-step map, also known as the Poincaré map. this is used to discretize system instead of time eliminating 3 state variables. The Poincare map relates the mid-stance velocity of the rimless wheel (the state) and the fixed torso angle per step (the control variable) to the mid-stance velocity at the next step (the new state). Using this map, we set up a Dynamic Programming problem to minimize a weighted sum of the square of the deviation from the desired speed and the actuator effort. The problem is then solved using a combination of value- and policy- iteration for computational efficiency. We demonstrate that it is possible to switch from a mid-stance speed of 2 $\frac{rad}{s}$ to 3$\frac{rad}{s}$ in 6 steps. Our results suggest that Poincaré map based Dynamic Programming is a computationally efficient.

In the model we use the parameters retrieved from the physical system measurements shown in Table 5.3. The Masses of all components were extracted from the Solidworks model and later verified using a scale. The torso length is the length from the axis of rotation at the hip to the torso's center of mass this result was also extracted from the cad file but later verified via knife edge balancing. The length of the wheel is measured from the hip to the unsprung foot. Between each simulation we changed β to produce different behaviors. The cost function in Equations 5.1

Parameter	Value
Torso Mass	4.194 kg
Torso Length	0.048 m
Wheel Mass	2.320 kg
Wheel Length	0.260 m

Table 5.3: Modeling parameters

is defined as the weighted sum between 2 terms. the first is difference between the angular speed of the leg taken at the poincaré section and the target mid-stance velocity. second is the normalized torso angle, a measurement of the actuator effort. using this cost function we consider 3 cases of different values for β. First β = zero, here there is no penalty set on actuator effort so more aggressive action is rewarded by way of lower cost. In the case of β = 1 both the state taken at the

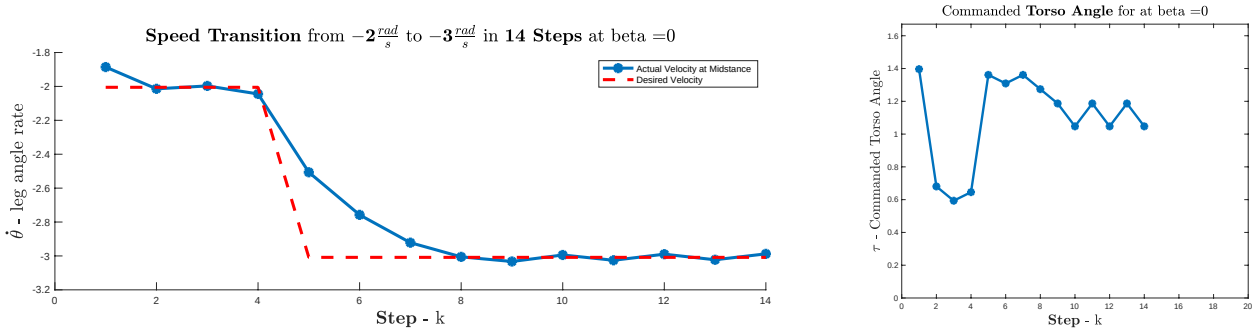

Figure 5.5: Control signal (right) and response (left) for Rowdy at $\beta = 0$

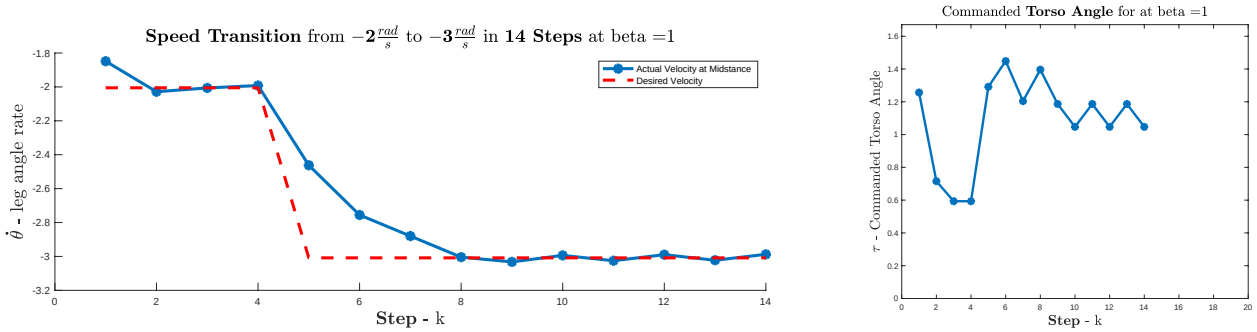

Figure 5.6: Control signal (right) and response (left) for Rowdy at $\beta = 1$

poincaré section and the actuator effort equally weighted forcing a less aggressive solution.

$$J = \sum_{k}^{n} \left(\left(\frac{\dot{\theta}_k - \dot{\theta}_d}{\dot{\theta}_{max}} \right)^2 + \beta \left(\frac{\phi_k}{\phi_{max}} \right)^2 \right) \tag{5.1}$$

5.2.1 Control and Response

The optimal control law was implemented in simulation of Rowdy using a look up method. The body angle of Rowdy is assumed to be exact in that when an angle is commanded the inverse kinematics is used to determine the forces necessary to hold the body steady at the commanded angle. there were three simulations done using optimal control laws from cost networks generated with three different values of β. Shown in the figures below are the commanded velocity, the torso angles found by Dynamic Programming to best produce those velocities, and the velocity at mid-stance response for values of β equal to {0,1}. Recorded in Tables 5.4 and 5.5 are the control signal (commanded torso angle) and the response when initially transition from a zero velocity

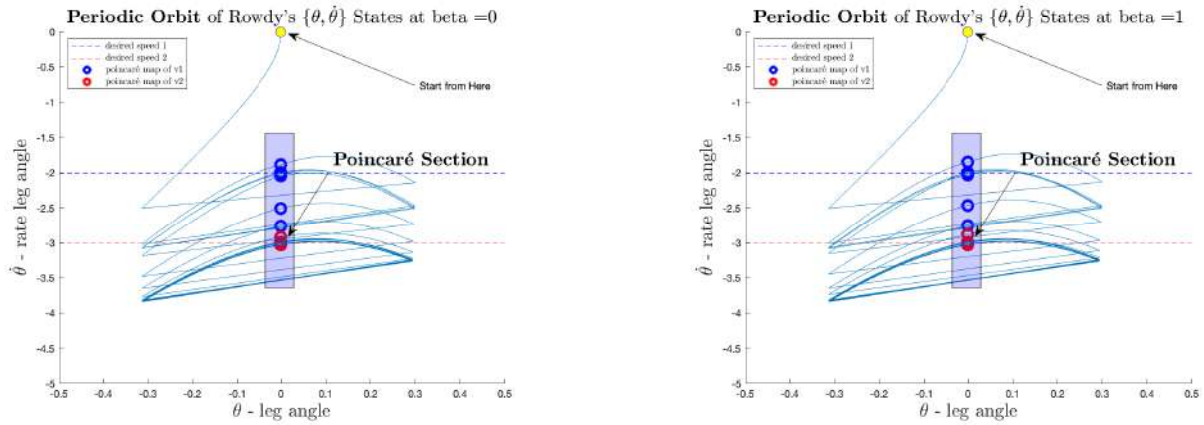

Figure 5.7: **Periodic orbit** of **rimless wheel state** for commanded speed transition from $-2.005\frac{rad}{s}$ to $-3.00\frac{rad}{s}$ at β = 0 (left) and β = 1 (right)

Value of β	0^{th}	1^{st}	2^{nd}
0	1.396	0.681	0.593
1	1.257	0.716	0.593
1e4	0.436	1.169	0.663

Table 5.4: Control signal from take off $0\frac{rad}{s}$ to steady state and -2.005$\frac{rad}{s}$

at launch first achieve a steady state at first desired speed. Recorded in Tables 5.6 and 5.7 are the control signal and the response when immediately after a desired speed change from $-2\frac{rad}{s}$ to $-3\frac{rad}{s}$ until steady state was achieved. Notice the less aggressive change in mid-stance velocity produced by the higher values of β.

Value of β	0^{th}	1^{st}	2^{nd}
0	-1.886	-2.014	-1.997
1	-1.848	-2.028	-2.006
1e4	-0.832	-1.931	-2.023

Table 5.5: State taken at poincare section from take off at 0 $\frac{rad}{s}$ to steady state and -2.005 $\frac{rad}{s}$

Value of β	0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}
0	0.646	1.396	1.396	1.396	1.396	1.396	1.396
1	0.593	1.396	1.396	1.396	1.396	1.396	1.396
1e4	0.593	1.396	1.396	1.396	1.396	1.396	1.396

Table 5.6: Control signal during speed change from $-2.005\frac{rad}{s}$ to $-3.51\frac{rad}{s}$

Value of β	0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}	6^{th}
0	-1.999	-2.045	-2.513	-2.779	-2.941	-3.043	-3.109
1	-2.016	-1.998	-2.488	-2.764	-2.932	-3.037	-3.105
1e4	-2.017	-1.999	-2.488	-2.764	-2.932	-3.037	-3.105

Table 5.7: State taken at poincare section during speed change from $-2.005\frac{rad}{s}$ to $-3.51\frac{rad}{s}$

CHAPTER 6: CONCLUSIONS & FUTURE DIRECTIONS

In conclusion, we developed a method to generate Dynamic Programming problems for physical systems and find optimal control policies quickly using parallel computing. This method was subsequently tested on two dynamically different system using different modeling approaches. One, using a full state approach and the other, using a reduced state approach. Both producing optimal policies driving the system to the goal state in a globally optimal manner as described by the cost function. The power of Dynamic Programming is displayed in its ability to create complex yet elegant nonlinear solutions to difficult problems. The robustness of Dynamic Programming seen in its ability to solve many problems with a single approach yet there exist a caveat: the Curse of Dimensionality.

6.1 Full State Low Resolution Approach

Dynamic Programming requires a disproportionally large amount of time to organize the problem and find optimal policies as complexity of the problem grows. There exist alternative algorithms which are significantly faster yet produce solutions comparable in performance. Many of these algorithms require some initial value or policy on which they can improve. In these cases low resolution Dynamic Programming considering the full state can be performed in a relatively short amount of time to produce an optimal policies given the resolution. This low resolution policy can then be refined at higher resolutions with faster algorithms to produce better performing policies at lower termporal cost.

6.2 Slow Decent Adaptive Model to Reduce Modeling Errors

Another issue in the application of Dynamic Programming policies, is that they are susceptible to modeling errors. In the case of the Rowdy Runner, modeling errors mostly present as steady state errors, sometimes driving the system to infeasalble states. To remedy this issue better models need to be created. There are two approaches to generating better models: either gather better data

and more of it, or use an adaptive model. Focusing on the adaptive model approach, Dynamic Programming is a continually updating process. In leu of this fact, it is suspected that a Dynamic Programming cost function that considers the error in the model for a model that changes at slow rate could be used to eliminate modeling errors. Dynamic Programming on a more accurate model would produce better performing policies in practice. One possible way to implement this approach would be develop an offline policy from the current model for the system to follow initially, then refine that policy using policy iteration online.

6.2.1 Launch Control

Currently Rowdy requires an initial push to start walking. This is due to the location of its center of mass. In the current design, Rowdy's centor of mass cannot be extended beyond the contact point at its leading foot when at rest. There are two possible solutions to this problem. The first, a redesign of the torso, redistributing the mass further from the axis of rotation. The second, rather than a design change, a programmatic solution would be to use Dynamic Programming considering all states of the system at rest to develop a launch controller. This controller would maximize the angular momentum in a specific direction considering the system as a simple pendulum and apply a counter torque to overcome the activation energy required to begin motion.

6.2.2 Stopping

The approach to variable speed control described in this document considers only the poincaré map of Rowdy. This map does not include a zero velocity state at mid-stance [9]. One possible solution is to use Dynamic Programming targeting the full state of the system to find the best input actions to produce a decaying limit cycle. Another possible solution is to apply torso angles to produce counter torques slowing the robot and apply a counter torque to the torso to suddenly reduce the moment.

BIBLIOGRAPHY

[1] Denardo E V. Dynamic programming: models and applications. Courier Corporation, 2012.

[2] Kaelbling L P. Value Iteration. https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node19.html.

[3] Howard R A. Dynamic Programming and Markov Processes. MIT Press, 1960.

[4] Dynamic Programming (ECO 10401 - 001), 08, 2014. http://ciep.itam.mx/~rtorres/progdin/.

[5] Kaelbling L P. Policy Iteration. https://www.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/node20.html.

[6] D O J M P. An Introduction to Dynamic Programming: The Theory of Multistage Decision Processes. Chapman and Hall LTD, 1967.

[7] Bellman R. Dynamic programming. Courier Corporation, 2013.

[8] Bellman R E, Dreyfus S E. Applied dynamic programming, volume 2050. Princeton university press, 2015.

[9] Mandersloot T, Wisse M, Atkeson C G. Controlling velocity in bipedal walking: A dynamic programming approach. in: Proceedings of Humanoid Robots, 2006 6th IEEE-RAS International Conference on. IEEE, 2006, 124–130.

[10] Bhounsule P A, Ameperosa E, Miller S, et al. Dead-beat control of walking for a torso-actuated rimless wheel using an event-based, discrete, linear controller. in: Proceedings of ASME 2016 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. American Society of Mechanical Engineers, 2016, V05AT07A042–V05AT07A042.

[11] Zamani A, Bhounsule P A. Foot placement and ankle push-off control for the orbital stabilization of bipedal robots. in: Proceedings of Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on. IEEE, 2017, 4883–4888.

[12] Bhounsule P A, Zamani A. A discrete control lyapunov function for exponential orbital stabilization of the simplest walker. Journal of Mechanisms and Robotics, 2017, 9(5):051011.

[13] Bhounsule P A, Zamani A, Pusey J. Switching between limit cycles in a model of running using exponentially stabilizing discrete control lyapunov function. in: Proceedings of 2018 Annual American Control Conference (ACC). IEEE, 2018, 3714–3719.

[14] McGeer T, et al. Passive dynamic walking. I. J. Robotic Res., 1990, 9(2):62–82.

[15] Tedrake R. The Simple Pendulum. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-832-underactuated-robotics-spring-2009/readings/MIT6_832s09_read_ch02.pdf.

[16] Atkeson C G, Liu C. Trajectory-based dynamic programming. in: Proceedings of Modeling, Simulation and Optimization of Bipedal Walking, pages 1–15. Springer, 2013.

[17] Lawrence S A. Basic Lagrangian Mechanics, 11, 2006. http://physicsinsights.org/lagrange_1.html.

[18] Hand L N. Analytical Mechanics. Cambridge University Press, 1998.

[19] Sundström O, Ambühl D, Guzzella L. On implementation of dynamic programming for optimal control problems with final state constraints. Oil & Gas Science and Technology–Revue de l'Institut Français du Pétrole, 2010, 65(1):91–102.

[20] Miller S, Ameperosa E, Seay K, et al. The Roadrunner: A 2-D Powered Rimless Wheel Robot for Energy-efficient and Rough Terrain Locomotion. Dynamic Walking, 2015.

[21] Garcia M, Chatterjee A, Ruina A, et al. The simplest walking model: stability, complexity, and scaling. Journal of biomechanical engineering, 1998, 120(2):281–288.

VITA

Robert Brothers, was born in San Antonio, Texas on November 2, 1989 to Howard and Annette Brothers. He was raised on a farm in Gonzales, Texas by parents who instilled in him a love of math and science. They provided support and encouragement throughout his academic career and nourished a curiosity that only grew with time. In the spring of 2015 he graduated with his Bachelor's of Science in Mechanical Engineering and a Bachelor's of Science in Chemistry.