May 2014

DARPA Robotics Challenge

Henrique A. Polido
Worcester Polytechnic Institute

DARPA Robotics Challenge

A Major Qualifying Project
Submitted to the Faculty of
Worcester Polytechnic Institute
in partial fulfilment of the requirements for the
Degree in Bachelor of Science
in
Mathematical Sciences and Computer Science
By

Henrique Polido

Date: May 1, 2014

Approved:

Professor B.S. Arthur C. Heinricher

Professor B.S. Michael A. Gennert

Keywords:
1. Inverse Kinematics
2. Robotics
3. Dynamic Programming

Abstract

This paper presents the Worcester Polytechnic Institute(WPI) and Carnegie Mellon University(CMU) team approach for competing on the DARPA Robotics Challenge (DRC) using the humanoid Boston Dynamics Atlas robot. An overview and analysis of the hardware and software architecture is described with emphasis on two of the challenges tasks, Wall and Drill. The realization of a double stance Inverse Kinematics(IK) full-body controller used for manipulation and its overall performance is elaborated. Moreover, an analysis for using Dynamic Programming optimization for robotic humanoid path planning is developed.

Acknowledgements

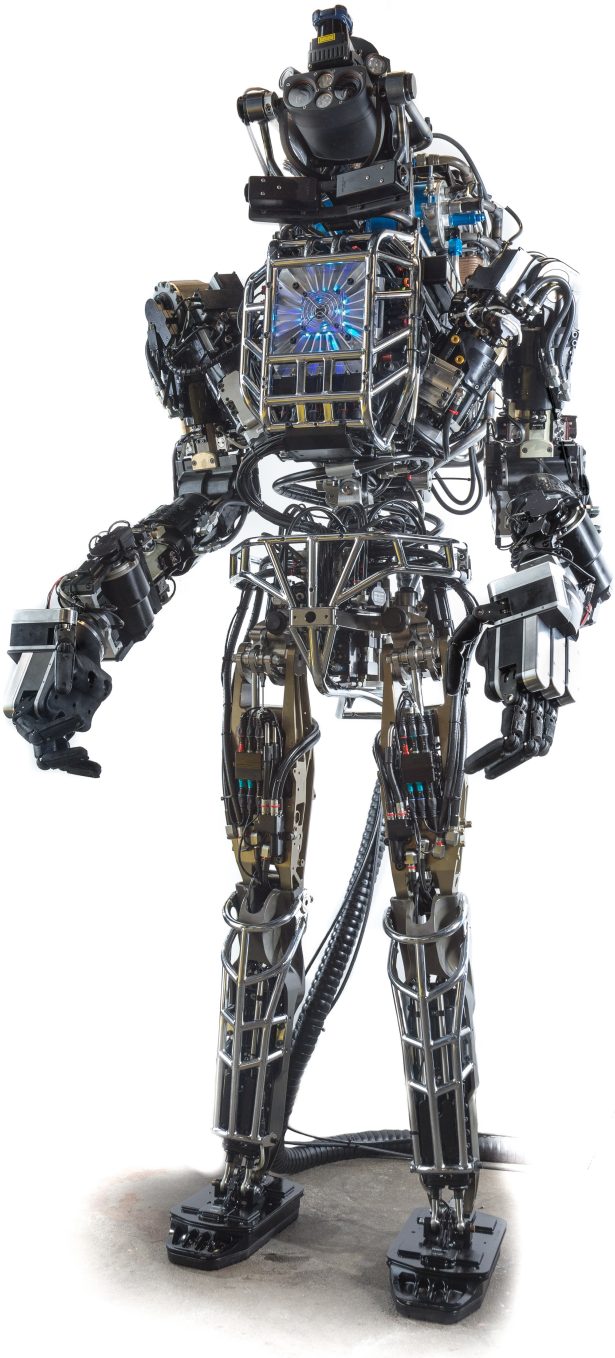
I would like to express my very great appreciation to my project advisers, Professor Arthur C. Heinricher and Professor Michael A. Gennert without whom this project would not have been possible. Their knowledge on the field and support to keep moving forward helped in the continuous development of the software for the Atlas robot.

Special thanks for Worcester Polytechnic Institutes, and the great support it has had on my education, and the influence on my career. Honourable mentions towards Mathew DeDonato, Velin Dimitrov, and Perry Franklin.

A huge thanks for my brother, Felipe Polido, to have introduced me into the project, and helped me with the many challenges of participating in a team, working through problems, and finding new solutions.

Thank you Carnegie Mellon University, with special consideration towards Eric Whitman, Chris Atkeson, and Siyuan Feng, for that without your help on the development on the controller, and many of the amazing motions that Atlas is able to achieve today.

Lastly, thanks to the Nvidia Corporation for providing Worcester Polytechnic Institute for top of line graphical processing units.



Contents

1 Introduction **1**

2 DARPA Robotics Challenge **3**
2.1 Hardware . . . 3
2.1.1 Atlas . . . 3
2.1.2 Sensors . . . 4
2.1.3 Robotiq Hand . . . 5
2.2 Software . . . 6
2.2.1 DBI Behaviours . . . 6
2.2.2 Architecture . . . 7
2.2.3 Network . . . 7
2.2.4 ROS . . . 8
2.3 Full-body controller . . . 8
2.3.1 SD/FAST . . . 9
2.3.2 Quadratic Programming . . . 9
2.3.3 Inverse Kinematics . . . 9
2.3.4 Inverse Dynamics . . . 11
2.3.5 Finding grasping positions . . . 12
2.3.6 Results . . . 12
2.3.7 Future Work . . . 12
2.4 DRC Valve . . . 13
2.4.1 Approach . . . 13
2.4.2 Results . . . 13
2.4.3 Conclusion and Future Work . . . 14
2.5 DRC Wall . . . 14
2.5.1 Approach . . . 14
2.5.2 Results . . . 15
2.5.3 Conclusion and Future Work . . . 15

3 Robust Dynamic Programming **16**
3.1 Looking at a Problem . . . 16
3.2 Running through an example . . . 18

3.3 Analysing this solution 19
3.4 CUDA 21
3.5 Applying DDP with CUDA 22
3.6 Results 23
3.7 Conclusion 23
3.8 Future Work 23

4 Future Work **24**

Chapter 1

Introduction

The work described in this report was done during the 2013-14 academic year in support of the WPIs participation in the DARPA Robotics Challenge (DRC). WPI participated in two teams for the competition held in Homestead, Florida in December 2013. For one of the teams, WPI faculty and staff joined a group headed by Drexel University. For the other team, WPI partnered with faculty, staff, and students from Carnegie Mellon University. The work described here was done in support of the WPI-CMU team.

There have been tremendous advances in solving the problem of controlling a humanoid robots [12]. Some of the mechanical problems have been solved by engineers at Boston Dynamics with the development of the Atlas robot. Key properties of the hardware, including sensors and actuators, are discussed in Section 3 below.

Advances in mathematical modelling for the control of robotic systems have also made significant advances in recent years. While the theory is well developed [6], limits in computational speed and power have kept the theory far short of practice. One of the standard approaches for solving the kind of optimization problems that arise in motion planning for a robot is the method of Dynamic Programming, originally developed by Richard Bellman. [6] While the theory is powerful, the approach suffers from the curse of dimensionality [1] and computational approaches fail for any realistically high-dimensional models. The Atlas robot is very high dimensional, 28 independent hydraulic actuated joints. In addition, computational approaches to optimization can be very sensitive to model and sensor error.

Any efficient computational approach must work with an approximate state space and an approximate controller and an approximation to the solution to the dynamic programming equations. Atkison's [1] and his co-authors have studied an approximation method that replaces a full optimization at each point in the state space with a simple comparison between the current best control strategy and a single, randomly generated, alternative. Whitman [12] used this approximation technique but also studies an approach that separates the high-dimensional control problem for the Atlas robot into a family of low-dimensional (and so feasible) optimization problems and builds an approximate solution by integrating information from the many separate problems. Part of the

work done in this project was on the implementation of approximation methods and techniques for speeding up the dynamic programming algorithm. The approach is applied to the simplest model problem: the swing-up pendulum. [12]

We will begin by describing some of the background, the motivation and the details in the Robotics Challenge, focusing on the properties and the limitations of the hardware. The next section considers the software, starting at the overall infrastructure, and going into more detail about the manipulation software.

Chapter 2

DARPA Robotics Challenge

The DARPA Robotics Challenge (DRC) is a competition of teams developing robots capable of assisting humans in responding to natural and man-made disasters. The goal of the DRC is to design a robust robotic system that has mobility and dexterity in disaster zones, the ability to use tools designed for humans, and supervised autonomy.

Teams are spread in several tracks. Worcester Polytechnic Institute is in the Track C team, in which competed in the Virtual Robotics Challenge (VRC), and received funding for the Atlas robot.

2.1 Hardware

2.1.1 Atlas

As described by the Atlas Robot Operation and Maintenance Manual, "The Atlas robots on-board features include a real time computer, a hydraulic power pack, force/torque sensors, inertial sensing, and active thermal management. In addition to the integrated sensors, the Atlas robot comes equipped with the MultiSense-SL sensor head from Carnegie Robotics, and modular wrists"

WARNER is 6 foot 2 inches tall and weighs approximately 330 pounds. The movement is powered by 28 hydraulically-actuated rotation joints, with a closed-loop position, velocity and force control.

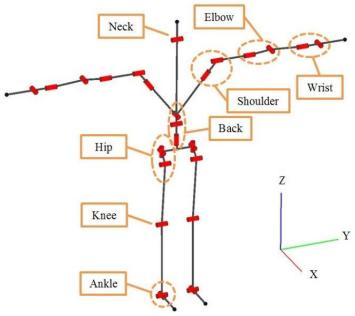


Figure 2.1: Atlas' set of joints

2.1.2 Sensors

Atlas is equipped with the following perception sensors:

Carnegie Robotics Sensor Head

The MultiSense SL is a tri-modal range sensor, providing laser range data, stereo range data, and video output stream. The device supply the majority of perceptual data used for tele-operation as well as automated control.

The apparatus brings together a Hokuyo UTM-30LX-EW laser, which output 43,000 points per second, and a Carnegie Robotics MultiSense S7 high resolution stereo camera, with a 4 megapixels on each side. On-board processing handles image rectification, stereo data processing, time synchronizing of laser data with a spindle encoder, spindle motor control, and lighting timing. [9]

Figure 2.2: Multisense SL

Situational awareness cameras

Two situational awareness cameras are placed just below the head of the robot, providing a wide range view of the workspace.

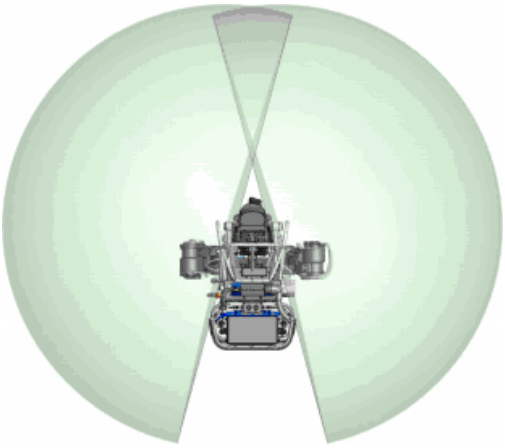


Figure 2.3: Top-down view of the robot, display range of view

Figure 2.4: View of the camera

Atlas Control Parameters

There exists several controller parameters that is hidden to the developer that controls the hydraulic joints. We let q, qd and f be the senses position, velocity, and torque respectively. And we let q_d, qd_d, and f_d be the desired position, velocity, and torque for each joint.

We create the following variables to help control the desired torque control:

- k_{q_p}: Position error gain, in $\frac{N \cdot m}{rad}$.
- k_{q_i}: Integral of position error gain, in $\frac{N \cdot m}{rad \cdot s}$.
- k_{qd_p}: Derivative error gain, in $\frac{N \cdot m}{\frac{rad}{s}}$.
- k_{f_p}: Proportional force feedback gain.
- ff_{qd}: Feedforward velocity gain.
- ff_{qd_d}: Feedforward desired velocity gain.
- ff_{f_d}: Feedforward desired force gain.
- ff_{const}: Constant force term.

And use them accordingly:

$$\begin{aligned} k_{q_p} \cdot (q_d - q) & + \\ k_{q_i} \cdot 1/s * (q_d - q) & + \\ k_{qd_p} \cdot (qd_d - qd) & + \\ k_{f_p} \cdot (f_d - f) & + \\ ff_{qd} \cdot qd & + \\ ff_{qd_d} \cdot qd_d & + \\ ff_{f_d} \cdot f_d & + ff_{const} \end{aligned}$$

2.1.3 Robotiq Hand

The robot gives the option of interchanging the hands. DARPA offered two primary options: the iRobot hands, and the Sandia hands. While both options are from known companies in the industry, neither provided the power, nor robustness necessary to complete the high demanding tasks of the DRC. These hands were designed to perform precision tasks, such as picking up a coin, or a fist-sized rock, but not to handle a power tool, or hold the Atlas robot while climbing up a ladder. The team chose an alternative end-effector developed by Robotiq, a 3-Finger adaptive industrial robot gripper, which provided a much stronger grip.

2.2 Software

The software developed my WPI and CMU are mostly written in C++, with small parts written in Python.

2.2.1 DBI Behaviours

Boston Dynamics provided predefined behaviours for Atlas that allow for basic performance like balancing and walking. Transitions between behaviours can occur manually or automatically, and follow a pre-determined state-machine.

Manipulate

In manipulation mode the user can control all the joints above the torso, and basic control of the pelvis orientation and height. The legs and overall balance are controlled by the Boston Dynamics controller embedded in the firmware.

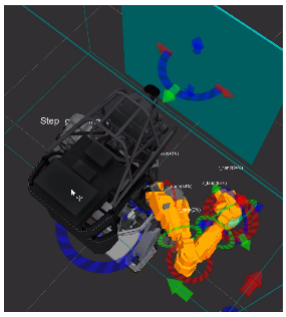


Figure 2.5: Rviz within ROS

The manipulation mode has a wide range of motion if the pelvis position and orientation is controlled together with an inverse kinematics solver for the joints, but can be unpredictable. The robot back joint that controls forwards and backwards motion was originally to weak, is unable to maintain position in certain configurations. The failure of this joint may compromise any given manipulation task.

Lastly, in this mode the user has the option to input any additional weight from objects in the hand to help the controller maintain balance.

Walk

Boston Dynamics walking behaviour works by the user providing a sequence of steps parameters, which include the step duration, sway duration, swing height, lift height, and step end distance.

The WPI team developed an additional interface for Rviz so that the robot operator is able to specify a final desired feet positions for the robot, and the software produces all the intermediate steps. It took some trial-and-error testing to figure out what the maximum step size is, and the program does not account for any debris on the floor.

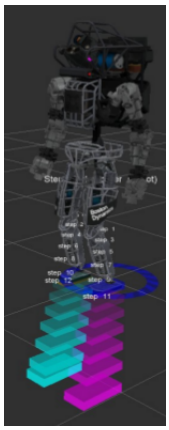


Figure 2.6: The planned steps for the robot walking forward and to the left

Although it could be automated, it is also usually required for the operator to additionally modify the final two steps such that they are further apart, thus providing a larger support polygon for manipulation.

2.2.2 Architecture

The system architecture was setup in three major components:

Atlas

The control loop inside the Atlas robot is inaccessible by the team, and can only be interfaced through a C++ API provided by Boston Dynamics. The API works by connecting to the robot through an IP address, queuing packets received from the robot, and sending back responses.

Field Computer

The Field Computer is designed to handle the team programmed controller logic of the robot, either being a full body controller, or using the predefined behaviours given by Boston Dynamics. This is also where high-bandwidth perception and autonomous algorithms function.

If the Field Computer fails to send a new command to the robot within 0.2 milliseconds, the robot will automatically shutdown.

Lastly, it is responsible for sending and receiving high level commands from the Operational Command Unit.

Operational Command Unit (OCU)

The Operational Command Unit(OCU) is the interface between the operator and the field computer. Our team relied on multiple operators working in individual OCU computers networked together. From the OCU the operators have to evaluate the robot situation, reason, and execute the desired tasks.

2.2.3 Network

DARPA structured a predictable, but limited bandwidth communication between the field computer and the OCU. The network quality varied every 60 seconds between a $1000\frac{kb}{s}$ with a $100ms$

round trip delay, and a $100\frac{kb}{s}$ with a $500ms$ round trip delay. This setup is to incentive the teams to perform the tasks autonomously.

The challenge used a Mini Maxwell [8], a network emulation system, to enforce the bandwidth limitations. The exact configurations of the Maxwell was never revealed, but whenever the bandwidth limitations was surpassed, packets seemed to be LIFO queued, sometimes causing images from the video stream to come out of order.

2.2.4 ROS

Robot Operational System (ROS) is a messaging messaging software that allows for remote procedure calls (RPC) through pre-defined message definitions. The basics of the framework allows for nodes to publish messages on given names, and any other piece of software on the network can listen to the message by knowing the name of published message.

For example, an important piece of software, TF, listens to messages about the robot joint states, and with additional information about kinematic model, publishing information about the transformation matrices between each frame of reference between on the body.

MoveIt!

MoveIt! is a software developed with ROS to help robot developers control the robot. It contains a inverse kinematics solver that can interface with Atlas, but prevents two containing limitations: *a*) The inverse kinematics solver has no option to give approximate solutions, and will fail to give joint state values unless a very approximate solution for a desired hand position exists. *b*) It does not account for modifying the pelvis position and orientation for solving for a solution.

2.3 Full-body controller

Carnegie Mellon University developed custom software for simultaneously controlling all joints of the robot as opposed to using one of Boston Dynamics's predefined behaviours. The CMU controller contains three major aspects, an Inverse Kinematics, an Inverse Dynamics, and a high-level controller.

Both Inverse Kinematics and Inverse Dynamics depends on two external libraries, SD/FAST and QuadProg++.

2.3.1 SD/FAST

The library SD/Fast provides physically-based simulation of mechanical systems by taking a short description of an articulated system of rigid bodies and deriving the full non-linear equations of the system. [11] The library takes several key factors about each sub-body in the robot, such as the link positions, mass, and inertia. The library than generates a C file that provides information such as the center of mass, Jacobian, and positions.

2.3.2 Quadratic Programming

Quadratic programming is a special type of mathematical optimization problem. It minimizes a quadratic function of several variables subject to linear constraints on the variables.

The problem is formulated as defining a square matrix Q, and column vectors, x and c, and minimizing the function, $f(x)$ in respects:

$$f(x) = \frac{1}{2}x^T Qx + c^T x \tag{2.1}$$

And have two types of constraints, of the form:

$$Ax \leq b \tag{2.2}$$

$$Ex = d \tag{2.3}$$

QuadProg++ is a quadratic programming solver. The algorithm implements the Goldfarb-Idnani active-set dual method. At present it is limited to the solution of strictly convex quadratic programs. [2]

The library has been rewritten using the Eigen datatypes, to utilize vectorization, and be easier to work with the rest of the code base.

2.3.3 Inverse Kinematics

Inverse kinematics is the process of given an desired effector positions, generate a joint state solution for the goal. There exists a great deal of research, and libraries that provide fast and concise solutions, but most of the research assumes that the base of the joint system is stationary, such as one that you might find on industrial manipulators.

Atlas is a bipedal robot, it is not simple to define a set of rules that describes all possible motions. Besides taking measure on the desired state, the kinematics also have account for the center of mass.

The robot is defined as tree/skeleton starting from the pelvis, defining the weight and distances between each joint. When we generate Jacobians [10] to move the end effector of each limbs, we also give it the ability to move the pelvis on a global coordinate frame. The hand Jacobians has no direct effect on the the leg joints, but indirectly moves the imaginary point that the pelvis should follow, and the legs move with it.

During this current iteration we have the following competing Jacobians that control the robot:

1. Left/Right hand position: Global desired Cartesian coordinates.
2. Left/Right hand orientation: Three local desired euler angles.
3. Torso orientation: Three local desired euler angles.
4. Pelvis orientation: Three local desired euler angles.
5. Center of Mass: Desired computed center of mass. Helps brings the robot back to a stable position.
6. Left/Right foot position: Global desired Cartesian coordinates.

The challenge of the system is to provide the correct weights for each the competing Jacobians such that the robot is able to perform the desired motions while keeping balance, and smooth motions. For example, one might decrease the weight of the center of mass, thus allowing the arms to reach farther out, but at the same time it would allow the robot to more easily enter undesired configurations that might cause a fall.

This system provides a know problem of singularities, in which when either a knee joint, or an elbow joint are straight in comparison to it's neighbouring links, the algorithm is unable to unlock them from their current state. The current solution is to simply prevent the joints to reach the singularities, simply by decreasing the joint limits.

Usually there are about 36 active Jacobians, there may be more or less depending on the required constraints, such as disabling a hand constraint, or enabling a desired elbow position constraint. The next procedure is an algorithm that is able to intake all the competing needs, and output the desired joint velocities.

The controller uses a system of calculating a pseudo-inverse, and finding a least squares solution by using quadratic programming. We first generate a matrix, A, of 34 columns and about 68 rows, and a vector, b, with the same name of rows. The matrix is composed of all the Jacobians multiplies by an weight, that must be equal to the desired change. An additional regularization identity element is added to help ensure that none of the joints move too quickly.

$$Ax = b \tag{2.4}$$

$$\begin{bmatrix} J_{Feet} & \cdot & W_{feet} \\ J_{Hand} & \cdot & W_{hand} \\ J_{Torso} & \cdot & W_{torso} \\ J_{Pelvis} & \cdot & W_{pelvis} \\ J_{CoM} & \cdot & W_{CoM} \\ I & \cdot & W_{regularization} \end{bmatrix} x = \begin{bmatrix} (Feet_d - Feet_a) & \cdot & Rate_{Feet} & \cdot & W_{feet} \\ (Hand_d - Hand_a) & \cdot & Rate_{Hand} & \cdot & W_{hand} \\ (Torso_d - Torso_a) & \cdot & Rate_{Torso} & \cdot & W_{torso} \\ (Pelvis_d - Pelvis_a) & \cdot & Rate_{Pelvis} & \cdot & W_{pelvis} \\ (CoM_d - CoM_a) & \cdot & Rate_{CoM} & \cdot & W_{CoM} \\ 0 \end{bmatrix} \tag{2.5}$$

We let $Q = A^t A$ and $c = A^t b$, and now we attempt to minimize for x:

$$f(x) = \frac{1}{2} x^t Q x + c^t x \tag{2.6}$$

With a joint limit inequality constraints, which prevents the solver, in a single time step, to set the joint values higher than the physical joint limits:

$$Wx < b \tag{2.7}$$

$$\begin{bmatrix} I & \cdot & timeStep \\ -I & \cdot & timeStep \end{bmatrix} x < \begin{bmatrix} -Joint_{Min} + Joint_{Actual} \\ Joint_{Max} - Joint_{Actual} \end{bmatrix} \tag{2.8}$$

QuadProg++ can now solve the problem, but there exists rare situations in which a solution is infeasible.

We now simply add the values of x into the current joint state of the robot, check again for any joints being out of bounds, and send the new desired joint position and velocities for the Inverse Dynamics.

2.3.4 Inverse Dynamics

Inverse Dynamics is responsible for generating the torques applied to each joint of the Atlas robot. The equations of motion and the constraint equations for a humanoid robot can be described as:

$$\begin{aligned} M(q)\ddot{q} + h(q,\dot{q}) &= S\tau + J^T(q)F \\ J(q)\ddot{q} + \dot{J}(q,\dot{q})\dot{q} &= \ddot{x} \end{aligned} \tag{2.9}$$

where $(q,\dot{q})$ is the state of the system, $M(q)$ is the inertia matrix, $h(q,\dot{q})$ is the sum of gravitational, centrifugal and Coriolis forces, S is a selection matrix, τ is a vector of joint torques, F is a vector of contact forces, and x is a vector of contact positions and orientations.

2.3.5 Finding grasping positions

On an attempt to perform the debris task more efficiently, a routine was created in which the program iterates forward several instances of the Inverse Kinematics part of the controller, and checks how well the controller is able to grab the target object, and picks the one in which the center of mass is most stable within the support polygon.

Several unexpected behaviours showed from this experiment: The controller would prefer to to leave joints at the joint limits unless it was complete necessary to move it to achieve a necessary motion, and when moving the arm away from the ground, the controller will choose to move the legs first until the legs reach a singularity.

It is unclear why the controller prefers to stay at joint limits, but might be due to the way that QuadProg++ adds special constraints while at a boundary, and does not move away from the edge cases unless necessary. Although no algorithmic solution was attempted, the current solution to not to allow the initial state when sampling goal positions to have have any joints at the limit.

Singularities are an inconvenient truth of working with Jacobians. The current solution is to simply avoid reaching singularities on the elbows and knees, or add special cases in the code. [12]

2.3.6 Results

Overall the controller is able to perform full body motions without much work from the developer, and sometimes act in an almost human-like behaviour. On the cases in which it was used, the full-body controller performed well on the DRC trials, and gave little trouble for the operator.

There still exists edge cases in which the controller will act unexpectedly, such as the elbows or knees reaching a singularity, joints being stuck at a boundary, and the controller performing large body motions for motions that could be accomplished only with the arms.

2.3.7 Future Work

Members at Carnegie Mellon University suggested to use SNOPT, a software package for solving large-scale optimization problems, to help with path planning. SNOPT is able to perform what QuadProg++ does, but over large motions and account for additional dynamic constraints.

Boston Dynamics has mentioned that the robot is receiving several major modifications on the arms and legs, including adding and switching the last 3 joints of each arm to be electric, instead of hydraulic, and providing new legs. All of these changes are bound to be completed by the end of January, which provides several complications in developing software that works specifically for Atlas in his current state.

2.4 DRC Valve

For this task the robot must close three industrial valves during a period of 30 minutes. Those consist of a 13” ball valve, and two gate valves with turning wheels of 9” and 18”. The ball valve has to be closed 90 degrees clockwise (until the lever is horizontal). The gate valves have to be rotated clockwise a full turn to be considered closed. [4]

2.4.1 Approach

The first problem to achieve the task is to step towards the valve. The operator must activate the point cloud feedback on the computer, and position a marker on top of what looks to be a valve on the collection of dots. With the marker in place, it gives the operator the position of where the feet must be placed wherever the robot is going to use the left hand, or the right hand, to operate the valve.

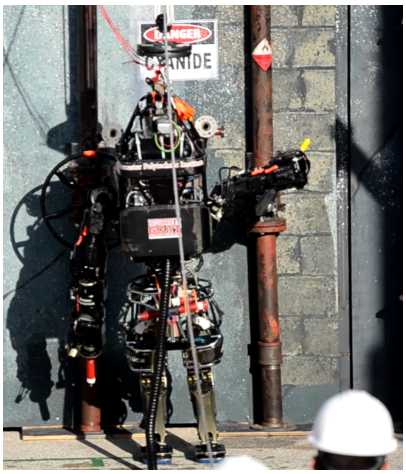


Figure 2.7: Robot in front of valves. Both shoulders right in front of each valve.

As demonstrated by a kinematic reachability study performed by MIT, the robot seems to have the best range of motion right in front of it’s shoulder. [5]

The initial approach to complete this task was to use the Boston Dynamics motion controller, and perform the motion by solving for different positions using MoveIt!, but this approach quickly presented several problems, such as when solving for the 6-DoF arm, it would only occasionally find a solution, and it was unable to complete the full motion. Another problem arose when using the 6-DoF arm + 3-DoF back joints, MoveIt! would use the back joints way too much for the motions and cause unpredictable behaviour. There was no direct way of limiting the motion of specific joints through MoveIt!.

The final method used the Carnigue Mellon’s Full-body controller. This allowed a human operator to choose in which quadrantal direction to move the end-effector, analyze the force being applied to the hand, and move the hand in which ever direction necessary.

2.4.2 Results

The task could be completed pretty robustly, and during the DRC trials, it was complete without any faults in 29 minutes, with only one minute to spare.

2.4.3 Conclusion and Future Work

It is feasible to perform the entire task autonomously. The valves are easy objects to recognize, since they are big extrusions from the wall, and the motions are fairly simple.

2.5 DRC Wall

For the Wall task the robot must use a single handed cordless rotary tool with an on/off switch to cut and remove a prescribed triangular shaped piece from a wall. The wall material will be 0.5 inch thick drywall. The three vertices of the right triangle will be 6 inches diameter circles. The edges connecting the vertices will be 6 inches wide. The vertical edge will be 12 inches long, and the horizontal edge will be 24 inches long. The bottom of the triangle will be 36 inches above the ground. [3]

Figure 2.8: The front camera vision of drilling the wall

2.5.1 Approach

Performing the Wall task was a bit of a challenge for two reason:
a) Being able to progress the power-tool trigger and *b*) having the mobility to cut the wall in one motion.

Picking up the drill

Finding the drill on the point cloud is mostly a trivial task for the robot operator. Specially since the hight of the table was given in the task description. On the robot walked to the table, DARPA gave the option to choose one of two drills:

- One Dewalt DCD980M2 cordless drill: The robot must grasp the drill and squeeze the trigger to maintain the drill on. The drill also has a side handle to help with maneuverability. The problem with this drill is that the Robotiq hand had no way to firmly grasp the handle and press the trigger at the same time. The shape of the handle would cause the hand's middle finger on the back side of the drill to push out the front finger out of the trigger whenever any kind of firm grasp was applied.
- Dewalt DC550 cordless drill: It has an on/off switch. It was quite easy to grasp and operate, but pressing the small button on the base of the drill was quite difficult with the robot's large hands. The most reliable solution that we found is to shake the drill after grasping it, and add an extension on the hand such that it presses on the ON button. Since the battery

asymmetrical, the heavier side always slowly moves towards the ground, and causes the ON button to fall onto the hand's added extrusion.

We found that the DC550 was the best option. The drill also provided a much wider range of motion while moving the arms.

Cutting the wall

Once the drill was turned on, the robot would walk towards the wall and position his right shoulder parallel to the bigger side of the triangle. Once the robot start cutting the wall, the two main issues were: *a*) Ensuring that the drill bit was far enough in the wall, and *b*) The battery of the drill was not drained.

DARPA promised the teams that the drill would be fully charged at the beginning of each run, but during one practice run before the competition, the battery ran out before finishing the first cut. The controller quite reliability able to perform the entire motion without having to take a step. The operator would tell the robot to move one centimeter at a time in a direction, and monitor the hand forces applied.

2.5.2 Results

During the competition, the operator made an error in the robot operation mode. Each of the hands has the option to either move the hand by setting a position relative to the floor, setting fixed joint values. While the right hand, which is operating the drill, is used with relative positions, the left hand is left at fixed joint values, not to add additional constraints to the kinematics.

The left hand was accidentally turned on into relative positions, and since there were no graphical feedback about the operation of each hand, the operator was unable to determine why the robot was unable to move as freely as necessary to cut the wall. On an attempt to reset the controller while the robot was still running to fix the unexpected behaviour, the robot fell, having to reset the task.

On the second attempt, the team only had about 10 minutes left to perform the task, and while it was possible to grab and turn on the drill, the robot was unable to finish the first cut in time.

2.5.3 Conclusion and Future Work

The approach of the task was solid and presented little issues most of the times. Due to the operator stress, and lack of visual cues about the robot state, the task was not completed. It is also feasible to perform the task autonomously with the current technology and resources.

Chapter 3

Robust Dynamic Programming

Model based optimal control is a powerful tool but relies heavily on the accuracy of the system model, which is not a trivial task to derive. There are many parameters involved, such as friction, sensor readings, calibrations, and the model may change over time due to wear, temperature, humidity, contact condition, and others.

Eric Whitman looks into using Multiple Model Dynamic (MMD) Programming, and uses the inverted pendulum swing-up problem to demonstrate the algorithm. [12]

By using the content provided in his thesis, the references, and example code, work on providing other students comprehensive guild on MMD for future students, we are looking at the different advantages, and shortcomings that comes with the algorithm.

Based on his previous work this paper aims to clarify the fundamental concepts of MMD and provide graspable examples. Furthermore, a faster running-time implementation using a General Purpose Graphical Processing Unit (GPGPU).

3.1 Looking at a Problem

To have a better understanding on how the algorithm works, we are going to look at the inverse pendulum swing-up example, which consists of a motor attached to the end of pendulum, and the controller aims to bring the pendulum to an upright position by applying torques based on the current state of the system.

The expected behaviour for the control law is to apply a torque equal to the direction the pendulum is swinging, such that more energy is added to the pendulum with each period, and when the pendulum finally has enough energy to perform a full rotation, apply a torque opposite to the movement such that the pendulum stops in an upright position.

Since our interested is in solving a variety of dynamic programming problems, we are going to

look at a more generic solution that discretizes the whole state of problem, and attempts to find a adequate policy to reach the desired goal.

We are going to assume that the pendulum has a length, $L = 1m$, and a weight, $m = 1kg$, and the world has a gravity $g = 9.81\frac{m}{s^2}$. The pendulum has a 2-dimensional state space, θ, the angle of the pendulum relative to the up position, and ω, the current velocity in radians per second. There is only a one-dimension action space, τ, the force of the motor can apply, and limit it at $\pm 1.5\frac{N}{s}$.

The pendulum dynamics can be described by:

$$\dot{\omega} = \frac{mLg \cdot sin(\theta) + \tau}{mL^2} \tag{3.1}$$

Define the state space as $x = \{\theta, \omega\}$, and the action space to be $u = \{\tau\}$. We define the pendulum to be pointing up at, $\theta = 0$, and we are optimizing the pendulum to reach $\theta = 0$, with zero velocity, $\omega = 0$, using the minimum torque necessary. We define the cost function be to L:

$$L(x,u) = \theta^2 + 0.5 \cdot \omega^2 + \tau^2 \tag{3.2}$$

And we want to minimize the total cost function, C, given by the integral:

$$C = \int L(x,u)dt \tag{3.3}$$

We discritize the state space of the pendulum by assuming $\theta \in (-\pi, \pi)$, and $\omega \in (-10, 10)$, and define a 2-dimensional increments of 0.016 radians, and 0.022 radians/second, giving us a table of 360,000 states. We also assume a time step of T=0.003 second delay between each action, and sensor reading of the current pendulum state. We define the iteration function:

$$x_{k+1}(u) = \begin{bmatrix} \theta_{k+1} \\ \omega_{k+1} \end{bmatrix} = \begin{bmatrix} \theta_k + T \cdot \omega_k + \dot{\omega} \cdot T^2 \\ \omega_{k+1} \end{bmatrix} \tag{3.4}$$

For each state of the pendulum, we define two elements, the value $V(x) = v$, and let the initial value of $V(x) = 0$ for all x, and the policy $U(x) = \tau$, and let the initial value for all states be equal to 0. For optimizing the policy, we guess a random torque, τ', for every state in the grid, x, and define:

$$V'(x) = C(x,u) + \gamma V(x_{k+1}(\{\tau'\})) \tag{3.5}$$

And if $V'(x) < C(x,u) + \gamma V(x_{k+1}(U(x)))$, we let $V'(x)$ and $U(x) = \tau'$.

We run the previous operation on all states of the grid for an uncertain number of times. For this particular program, running this optimization for 1000 iterations is sufficient for making the

pendulum converge to the desired goal state, but it can also be noted that running for 10000 times generates a more torque efficient solution.

The problem is simple enough with the single-link pendulum case, but when extra links are added to the pendulum, such as the double-link pendulum, the problem becomes increasingly difficult to solve, but the algorithm can still find a sub-optimal policy with the modifications of redefining the state space, action space, and the iterate function.

3.2 Running through an example

To better comprehend how this problem is solved, lets take a first iteration from a single discrete point in the grid. Suppose we are analyzing the $\omega_0 = \frac{\pi}{2}, \omega_0 = 1 rad/sec$, and since we are in the first iteration, our current policy is $\tau_0 = 0$. If we keep the current torque, we find that the next step is:

$$\dot{\omega_0} = \frac{1 \cdot 1 \cdot 9.81 \cdot sin(\frac{\pi}{2}) + 0}{1 \cdot 1^2} = 9.81 \tag{3.6}$$

$$\begin{bmatrix} \theta_{k+1} \\ \omega_{k+1} \end{bmatrix} = \begin{bmatrix} \theta_k + T \cdot \omega_k + \dot{\omega} \cdot T^2 \\ \omega_{k+1} \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} + 1 \cdot 0.003 + 0.5 \cdot 9.81 \cdot 0.0032 \\ 1 + 9.81 \cdot 0.003 \end{bmatrix} = \begin{bmatrix} \frac{\pi}{2} + 0.003044145 \\ 1.02943 \end{bmatrix} \tag{3.7}$$

And we can now discover the current value of $V(x)$ to be:

$$V(x) = C(x,u) + \omega V(x_{k+1}(\{\tau'\}) = (\theta^2 + 0.5 \cdot \omega^2 + \tau^2) + 1 \cdot 0 = 2.6032705 \tag{3.8}$$

Since this is the first iteration of this equation, the second term of the value equation is 0. Now suppose that we guess $\tau = 1$, and find the new state to be:

$$x'_1 = \begin{bmatrix} \frac{\pi}{2} + 0.003048645 \\ 1.03243 \end{bmatrix} \tag{3.9}$$

We find the new value of

$$V'(x) = 2.6062750 \tag{3.10}$$

Since $2.6032705 < 2.6062750$, we learn that we are applying a torque in the wrong direction, and it is better to let gravity pull on the pendulum instead of applying addition torque, and keep the value, $U(x) = 0$. Now suppose we guess $\tau = -1$, we find that

$$x'_1 = \begin{bmatrix} \frac{\pi}{2} + 0.003039645 \\ 1.02643 \end{bmatrix} \tag{3.11}$$

With the value

$$V'(x) = 2.6002660 \tag{3.12}$$

Since $2.6032705 > 2.6002660$, we change the value $U(x_0) = -1$. Giving a new optimal torque to apply at that point.

It is important to notice the optimal torque may completely change directions throughout the iterations due to the value of the neighbours changing. (Further details with the video)

3.3 Analysing this solution

The algorithm is far from being real time, since for the single-link pendulum case, on an 8-core CPU at 2.8 GHz, takes about 4 seconds for each iteration, and about 1 hour for the 1000 iterations, but it does generate very robust policy tables.

On the following plot we show the policy of 10,000 iterations running against a pendulum, position of the pendulum relative to the up position, starting at pi, pointing down, and with 0 velocity. It is interesting to see that the policy follows similar to a bang-bang controller, using near maximum torque on both directions, until the system has enough energy to reach the necessary solution.

The following plot demonstrates the value and the policy values of the pendulum swing-up solution. The X-axis is the velocity of the pendulum, while the Y-axis is the current position of the pendulum.

Figure 3.1 states that as the pendulum starts at $[\pi, 0]$, starting with the pointing down, with no velocity, and on the highest value on the value plot. Over time, the pendulum will travel down the apply torque and gain velocity over time, up to the point that the pendulum will reach the lowest value on the plot.

Figure 3.2 shows the position, velocity, and torque being applied at any moment on the pendulum. Even with 1,000 iterations the system was able to find a method in which the position converges to 0, when the pendulum is pointing up, but one can also see that during the last 20 seconds, the line contains quite some noise.

Lastly, on Figure 3.3 shows the paths of pendulum. Each red line represents the start configuration of a pendulum, and stops when they go outside of the plot, or reaches the desired position at $[0, 0]$. It was surprising to learn that on the top-right corner, when the pendulum has some momentum and is pointing down, they do not all converge exactly to the same path, but are rather somewhat parallel to one another.

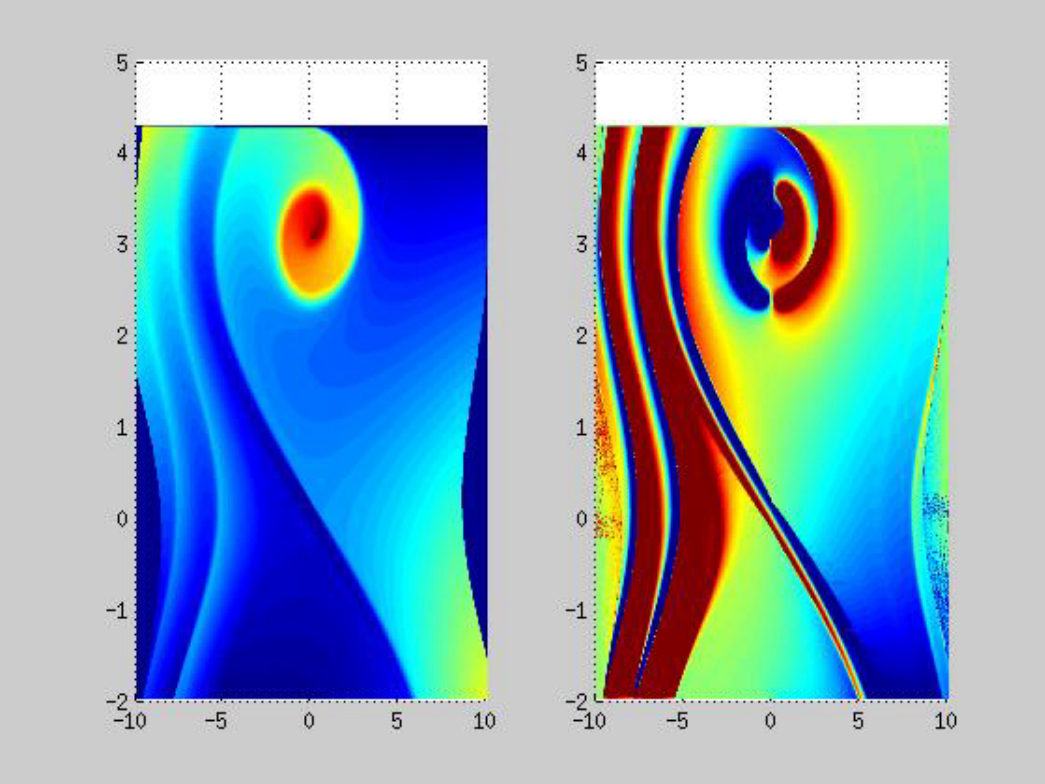


Figure 3.1: Plot showing the value and policy of a pendulum swing up

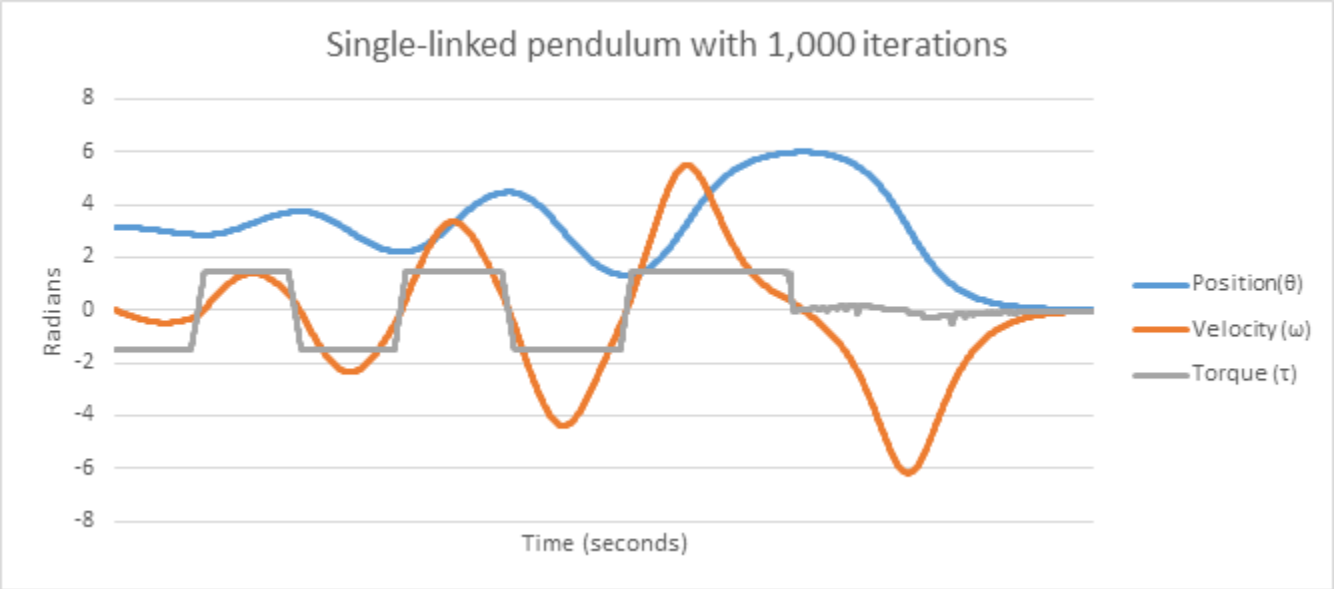


Figure 3.2: Position, velocity, and torque of pendulum over time

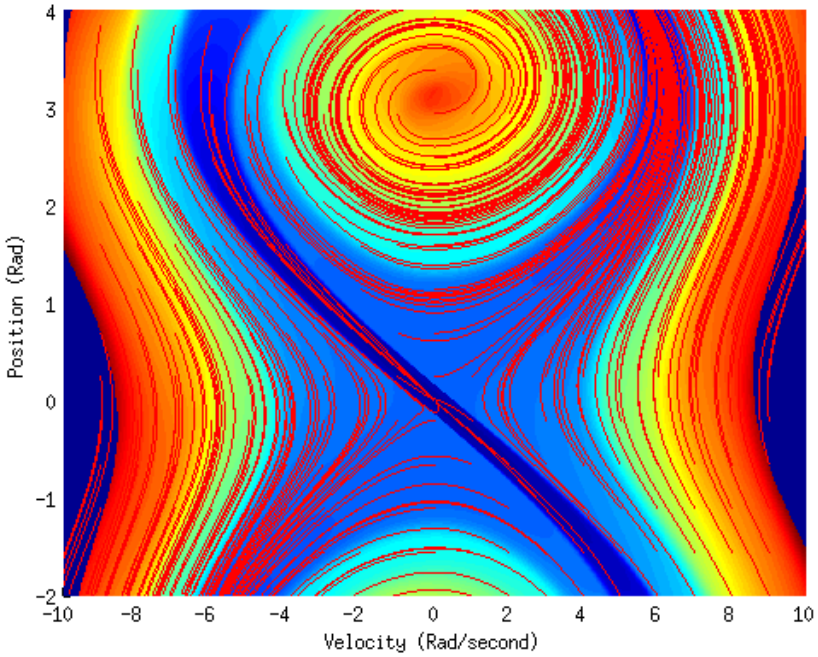


Figure 3.3: Paths that pendulum would follow given any start position

3.4 CUDA

To first understand why using a GPGPU is a faster tool that fits to solve this problem, we must first take a look at the history of how games and GPUs, and how the computing model of GPUs fits this problem.

The principles of a modern game rendering have not changed much since the beginning; A list of triangles in 3d space are transformed, projected into a plane, and then each pixel is rendered with the color of the closest triangle. The three major steps in the rendering pipeline involves:

1. Vertex processing: Each vertex in a model is moved into a new position by transformation matrices and weights.
2. Rasterization: Processing each of the triangles in the screen region, and calculating what the closest triangle in each pixel is.
3. Fragment shading: Determined the color of each pixel based on the closest triangle. (The coordinates of the closest triangle is passed into the fragment shader)

While the second step is usually done by hardware, hidden away from developers, the main aspects of the first and third step is that each vertex and pixel can be completely independent of one another and can be programmed by C-like programs, called shaders. Due the popularity and profitability

of game companies, this gave incentives for companies to develop dedicated hardware that process each of the points as quickly as possible.

Until 2001 developers could only use shaders that were pre-programmed into the hardware, and then Direct3D 8 [13] gave developers the option to program custom shaders that run in the hardware.

The main drive for GPU development was for computer graphics, but using some technique, one could speed up large linear algebra processing using storage through image textures. [7]

For example, OpenGL could be used to perform the multiplication of two 256 by 256 matrices. In this case, three texture buffers would be created inside of the GPU, each of the size of the matrix, and two of them would be loaded with the read-only input matrices values from the main computer memory, while the other one would be used as a render texture. When the fragment shader runs over all the pixels of the render texture, the user defined program would read the necessary values from the other two matrices and render the corresponding output value.

Later during 2007, NVIDIA released CUDA, a general purpose computing standard that would allow developers to create GPU based software, and later on 2008, the open source community followed with OpenCL. While OpenCL and CUDA follows the same principles as fragment shaders, it also opened doors for other applications, such as atomic operations, thread syncing, memory barriers, and others.

3.5 Applying DDP with CUDA

Since a new improved torques can be found in parallel with other points, we can run the Dynamic Programming Algorithm using CUDA. We initialize the process by allocating two memory buffers in the device, and reset all of the values to 0. We create a CUDA kernel, which takes an identification number, input memory buffer, and compares the current policy with a new random policy, and stores the result on an output buffer.

We then swap the input buffer with the output buffer and repeat the process for the given number of iterations. The process otherwise is equivalent to the CPUs counterpart, but instead of taking up to 1 hour to complete, the GPU process only took 2 minutes.

The limitations with using a GPU for this kind of process is a more limited memory space, while it can be relatively cheap to buy more computer RAM, or allow the computer swap memory into a solid state drive, the GPU is limited to only to the memory available on the device, and even devices top of the line today, such as the NVIDIA Tegra 4, only provides 4GB of memory.

3.6 Results

The results of the algorithm were never applied in a real system, and the generated value and policy for the GPU and CPU were essentially identical. Presenting sensor noise and other real-life factors into a simulation, we can still see that the system can still perform under inaccurate actuation. [12]

3.7 Conclusion

Dynamic Programming still has the Curse of Dimensionality, the dimensions of the project can be scaled down to fit into a manageable set, and since the data storage is rather comparably cheap, the information could be stored in disk for later retrieval. Furthermore, with the help of CUDA, specific sections of the grid could be recalculated with finer detail in real time when needed.

The Pendulum-Swing problem has been solved many times before, but with this type of solution, the problem can be extended to higher dimensions, and it only requires an linearised model of the system. The algorithm could be easily extended to take a double, triple, or any given number of links pendulum.

3.8 Future Work

This problem could be extended into looking at different end-effector positions for Atlas. One could generate a table of all feasible end-effector positions, and find paths on how to travel from one position to another while minimizing change in the center of mass deviation from the support polygon.

Chapter 4

Future Work

The WPI-CMU team performed well on the DRC trials. Regardless, due to the time constraints a lot of the code was built in a hurry, and without a proper structure. It is important to create a strong code base that can evolve over the years, and adapt to future challenges. An important aspect of having a strong base is being able to perform experiments, and build upon previously developed work. A lot of emphasizes after the DRC trials was directed on refactoring the code base, and there is still a long way to go.

As mentioned earlier in the paper, there is plenty of research in the subjects of object detection, manipulability, planning, and others, but it is a different challenge to be able to adapt all this work into a single code base, and be able to perform experiments, while using data from all the different aspects of the robot.

New goals for the team should focus on tools that can be reused and extended at later times. For example, with the step planner, a developer should be able to instantiate a new planner either with simulation, or the real robot, and have access to modify each of the steps. At a later time, a team can extend the initial step planner to automatically generate all of the steps, but should still leave the option to use the old planner.

A great library is one that allows the user to choose the tools to complete their task, and modify one of the steps as necessary, and it is a difficulty that research does not often present to students, since students are usually only required to use a library, instead of developing on the library itself.

The ultimate goal of this project would be to perform all the tasks a human is able to do, but developers must be able to break down the complex tasks into simpler steps, and understand the difficulties that comes before moving into solving a generalized case.

Bibliography

[1] Christopher G. Atkeson and Benjamin J. Stephens. Random sampling of states in dynamic programming. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICSPART B: CYBERNETICS*, AUGUST 2008.

[2] digaspero. Quadprog++, 2008. [Online; accessed 01-April-2014].

[3] Velin Dimitrov and Felipe Polido. Cyber physical development of tool grasping and manipulation using the atlas humanoid robot, 2014.

[4] Velin Dimitrov and Felipe Polido. Valve task, 2014.

[5] Maurice Fallon. An architecture for online affordance-based perception and whole-body planning, 2014. [Online; accessed 01-April-2014].

[6] R. W. Fleming, W. H. Rishel. *Deterministic and Stochastic Optimal Control*. 1975.

[7] Dominik Göddeke. GPGPU–Basic math tutorial. Technical report, Fachbereich Mathematik, Universität Dortmund, November 2005. Ergebnisberichte des Instituts für Angewandte Mathematik, Nummer 300.

[8] InterWorking Labs. Mini maxwell network emulation system, 2013. [Online; accessed 05-April-2014].

[9] Carnegie Robotics LLC. Mini maxwell network emulation system, 2014. [Online; accessed 05-April-2014].

[10] M. Vidyasagar Mark W. Spong, Seth Hutchinson. *Robot Modeling and Control*. 2006.

[11] Michael Sherman and Dan Rosenthal. Sd/fast, 2001. [Online; accessed 01-April-2014].

[12] Eric C. Whitman. *Coordination of Multiple Dynamic Programming Policies for Control of Bipedal Walking*. PhD thesis, Carnegie Mellon University, 2013.

[13] Wikipedia. Direct3d — Wikipedia, the free encyclopedia, 2014. [Online; accessed 01-April-2014].