April 2016

Control of Humanoid Robots for Use in Unstructured Environments

Perry Carl Franklin
Worcester Polytechnic Institute

Control of Humanoid Robots for Use in Unstructured Environments

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree in Bachelor of Science

in

Robotics Engineering

By

____

Perry Franklin

Date: 4/27/2016

Project Advisors:

____

Professor Taskin Padir, Advisor

Abstract

Humanoid robots have the potential to replace human beings for dangerous tasks, such as disaster relief. One of the most important abilities for a humanoid robot is the ability to manipulate its surroundings. We developed human-in-the-loop techniques for the Atlas platform to compete in the DARPA Robotics Challenge. Many of the tasks in the event required actuation of the environment, such as turning a valve, pulling a lever, and opening a door. This paper will detail our work on manipulation for humanoid robots. In particular, we will discuss our approaches to effective operator interface design, manipulation techniques and motion planning.

Acknowledgements

The author would like to thank all the members of the WPI-CMU DRC team, formerly the WPI Robotics Engineering C-Squad (WRECS) team.

Principal Investigators

- Professor Michael Gennert (WPI)
- Professor Taskin Padir (WPI) (Co-PI)
- Professor Chris Atekson (CMU) (Co-PI)

Engineers

- Mathew DeDonato (Team Leader)
- Kevin Knoedler
- Felipe Polido (Manipulation Leader)
- Doug Conn

Post-Doc

- Xiaopeng Chen (CMU)
- Chenggang Liu (CMU)
- Eric Whitman (CMU)
- Joohyung Kim (CMU)
- Weiwei Huang (CMU)
- Hirotaka Moriguchi (CMU)

Student Members

- Benzun Pious Wisely Babu (WPI) (Perception Leader)
- Josh Graff (WPI)
- Nandan Banerjee (WPI)
- Ruixiang Du (WPI)
- Peng He (WPI)
- Xianchao Long (WPI)
- Siyuan Feng (CMU)
- Ben Xinjilefu (CMU)
- Bowei Tang (CMU)
- Chris Bove (WPI)
- Aaron Jaeger (WPI)
- Lening Li (WPI)
- Xiongyi Cui (WPI)
- Daniel Miller (WPI)
- Alison Kristoff (WPI)
- Trent Tabor (WPI)
- Velin Dimitrov (WPI)

- Dongil Choi (WPI)
- Ryan Giovacchini (WPI)
- Nick Cyganski (WPI)
- Henrique Polido (WPI)

In particular, the author would like to single out Felipe Polido, Eric Whitman, Siyuan Feng, Xianchao Long, and Henrique Polido for contributing heavily to the manipulation technique described in this report.

The author would also like to thank Chris Atkeson (CMU) for allowing the author to visit CMU and participate in research on fall recovery. In addition, the author would like to thank Akshara Rai (Chris' PhD student) and Zhibin "Alex" Li (PhD) for working with the author, and allowing him to contribute to their work on fall recovery.

A thank you to John Schulman (UC Berkeley) who developed Trajopt, the motion planner we used, and who assisted us in implementing it on Atlas.

Finally, a thank you to Jerry Jaeger for providing photography. Photos taken by Mr. Jaeger are labeled with *PhotosbyJaeger*

Contents

Table of Figures:

1 Introduction

Humanoid robotics is an exciting field dealing with robots that superficially resemble humans. These robots offer unique advantages over other design types. Since they are shaped like humans, they are already adapted for environments designed for humans. As legged robots, they have incredible mobility; they can move in any direction from a standing position, and can traverse rough terrain. Finally, they have a very small workspace footprint. In other words, they can work in tight areas.

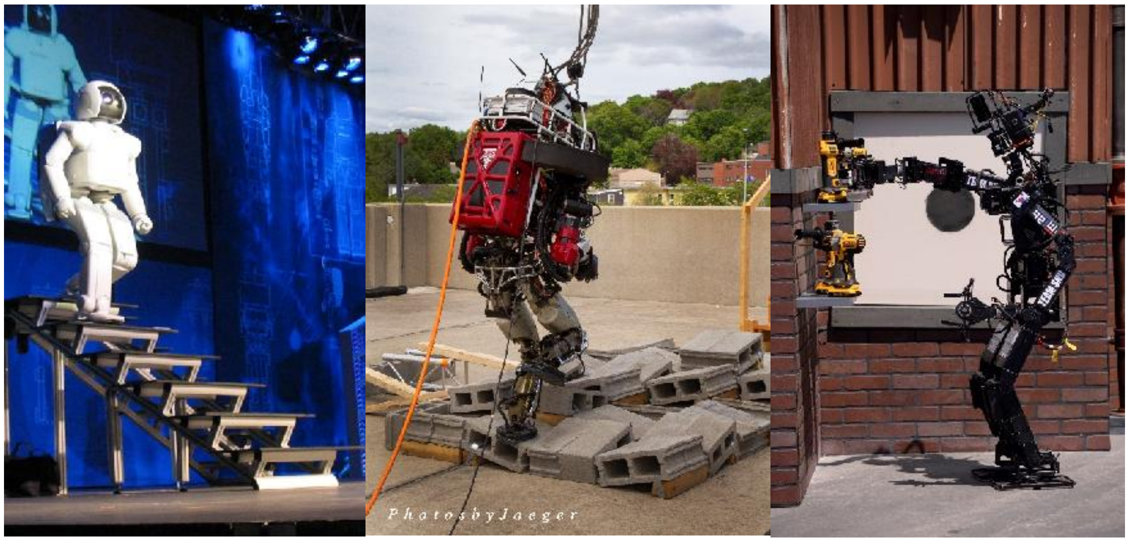


Figure 1: Examples of modern humanoid robots. Left: Honda's Asimo climbing down stairs. Center: Boston Dynamic's Atlas navigating rough terrain. Right: SNU's THOR operating a power tool (photo credit: DARPA).

However, humanoids come with the challenge of balance; that is, keeping the robot upright and on its feet. In addition, the calculations required to compute solutions to various problems are exponentially more complex due to the large number of joints present in humanoids. For example, a mobile base that uses wheels to drive will only need to compute motions for a few wheels for locomotion on a plane. A humanoid moving on the same plane will need to solve joint positions for at least 12 leg joints for each step. The balance issue also makes interacting with the environment more difficult for humanoids when compared to, for example, industrial robot arms. This is primarily due to the lack of a passively stable base – every action has the potential to cause the robot to lose balance and fall.

This Major Qualifying Project (MQP) developed techniques for humanoids to perform manipulation tasks. Manipulation refers to a robot's ability to interact with its environment. For instance, a robot may need to pick up a power tool or turn a valve.

Many robots that perform manipulation operate in structured, controlled environments. Industrial robots are excellent examples of this; their environments and tasks are so well controlled that they can repeat the same motion over and over again for the exact same result. The counterpart to the structured environment is the unstructured environment. In an unstructured environment, very little is controlled. Outdoor tasks are good examples of this; the terrain can vary wildly, the presence of wind can interfere with the robot's operation, obstacles can change location, and the lighting conditions can change unpredictably, just to name a few variables. These variations make unstructured environments difficult to work in.

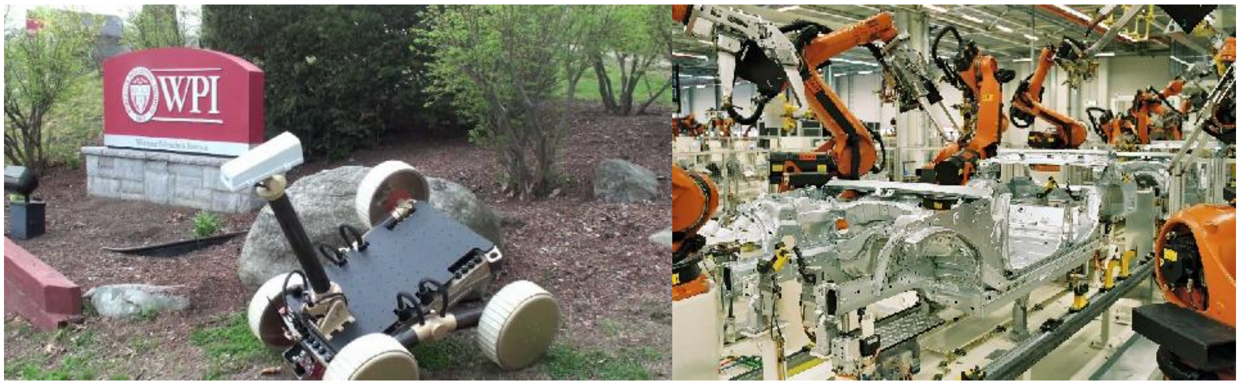


Figure 2: Examples of other robot designs besides humanoid. Left: The WPI Oryx rover, a mobile robot. It does not have to recalculate leg positions or maintain balance as it moves through its environment. Right: Kuka Industrial arms in an automotive factory. Note how the arms are all mounted firmly to the ground, and how the objects they interact with are constrained to designated paths.

Nonetheless, mastering unstructured environments is very important for increasing the utility of robotics. Humans live and work in unstructured environments; if robots are ever meant to work alongside human beings, unstructured environments must be conquered.

This project aimed to develop manipulation techniques for humanoids in unstructured environments. The project was prompted by the DARPA Robotics Challenge (DRC), which placed robots in a simulated disaster environment with a number of pre-defined tasks. The challenge took place outdoors.

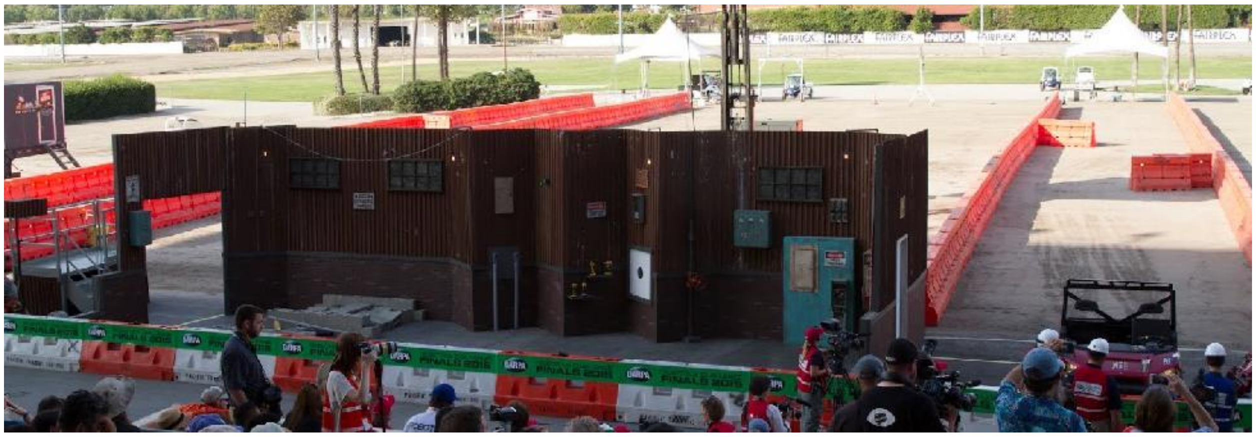


Figure 3: The DRC course. The course is run from right to left. First is the driving portion, followed by an "indoors" portion. The entire course is outdoors and in front of a live audience. Image credit: Jerry Jaeger.

The DRC also allowed for human operators. This made the problem significantly easier; analyzing the environment and searching for objects of interest and goals could be offloaded to the human operator, rather than developing perception algorithms to interpret the incoming sensor data. The manipulation techniques and methods developed for this project were designed for human operators. All of the manipulation techniques designed for the DRC were implemented on the Boston Dynamics Atlas robot.

This report will first go over some background information. This is organized into generic algorithms that can be used for many different robots and tasks, followed by the specific information for the robots worked on during this project. The report will then detail work done on user interfaces for manipulation; specifically, it will explain initial GUI design, subsequent lessons learned, and the final design. Following that, the report will discuss the motion generation techniques used by the WPI-CMU DRC team. The integration of these components to the overall system will be explained, followed by some results from the DRC. The report will finish suggestions for future work.

1.1 Authorship and Individual Contributions

Although this report only features one author, the methods for manipulation described within are by no means solely the author's. For any work performed on Atlas, credit is due to the WPI-CMU DRC team. This is not only for support and advice, but for development as well.

The major contributions of the author were to the GUI design and the integration of Trajopt with the rest of the system. In particular, the initial design for the manipulation interface, CMUManipulation, was developed almost entirely by the author. In the subsequent designs, the author had small influences on the individual task panels, but made major contributions to the dedicated manipulation Rviz plugin, CMUPreview. For the integration of Trajopt, the author made major contributions to the code base that allowed communication with the rest of the system. Small contributions were made to the mathematical development of constraints; however, the author was greatly involved in implemented the constraints and designing the overall optimization problem.

In addition to these specific contributions, the author was a general member of the team from the beginning of the DRC to the end. Throughout that time, he provided bug fixes, code improvements, and advice in general to the team.

2 Background

This focus of this project was on manipulation. However, for a humanoid robot to function at all requires a great deal of infrastructure, regardless of the tasks being performed. These range from general algorithms, such as state estimation and localization, to humanoid specific algorithms, such as the balancing controller.

This section will describe the prior research done for these algorithms, starting with general algorithms that are not specific to our robot. After that, specific algorithms and systems used on our Atlas will be explained.

2.1 General System Algorithms

In the field of robotics, there are many generic algorithms and common definitions. To facilitate the reading of this paper, this section will explain several of these that are pertinent to the project.

2.1.1 Task space vs Configuration space

For robotics, there are generally two different mathematical spaces that are relevant. These are task space and configuration space.

Task space refers to mathematical representations of the world. These take the form of coordinates and rotations that place an object in the world. For instance, an object like a lever or valve has a location in the real world, which is specified by its translational and rotational coordinates. A robot always has a representation in task space, which corresponds to the locations of points on the robot such as the location of the hands or feet. Since task space requires coordinates for both translation and rotation, it has 6 dimensions.

Configuration space encompasses all possible states of the robot. It is dependent on the number of degrees of freedom (DoF) on the robot, and additionally on the number of derivatives that are desired. For example, Atlas has 30 joints and has a mobile base that can move in 6 dimensions. This gives a configuration space with 36 dimensions. However, this only specifies the position of the robot and all the joints, and says nothing about higher derivatives such as velocity and acceleration. The configuration space can be extended to include these if necessary. In this report, configuration space will refer to configuration spaces with positions only.

It is possible to transform between these two spaces, but can be computationally prohibitive. The calculation to transform joint positions in configuration space to link positions in task space is called forward kinematics. The calculation to transform link positions in task space to joint positions in configuration space is called inverse kinematics, and is generally more difficult given the existence of multiple solutions or no solution. In addition, objects in task space can be transformed into configuration space. However, this calculation is very difficult to perform, and lacks a closed-form solution in most cases. When this calculation is required, it is performed on individual points.

2.1.2 State Estimation

State estimation is an important part of the robotics problem. It describes how to interpret sensor data, which can be noisy and not consistent. State estimation seeks to make the best estimate of the robots true state from this sensor data.

2.1.2.1 Raw Sensor Data

Humanoid robots are often equipped with multiple sensor inputs. Position sensors such as potentiometers and encoders are the bare minimum. These sensors measure joint angles, with a sensor located on each joint. Similarly, many robots have force sensors of some kind located on each joint. Another common sensor is the Inertial Measurement Unit (IMU), which is a combination accelerometer and gyroscope, and accordingly measures accelerations and rotation velocity. Force-Torque sensors are also common, located on points were contact is expected (such as feet or hands) and are used to measure reaction forces with the environment. Vision and Lidar systems are somewhat common, but involve complex algorithms for appropriately interpreting incoming data; while both robots used in the report are equipped with stereographic cameras and Atlas also has a spinning Lidar, this report will not cover methods that use this data for state estimation.

In simulation, direct use of this sensor data is very accurate and reliable as the model of the robot is mathematically reliable and precise. However, in the actual hardware many factors can lead to this data being unreliable and even unusable. Most sensors will contain some amount of noise in their outputs. Accelerometers are particularly notorious for this shortfall, as noise can be particularly crippling during integration (eg integrating from measured accelerations to velocities to positions). Force-Torque sensors often also contain large amounts of noise. In addition, unknown biases can cause errors in sensor readings, and the lag time between actual phenomenon and the sensor readings thereof can cause misinterpretation of the data.

Position sensors such as potentiometers and encoders have relatively low noise levels, and are usually the most precise sensors on a given robot. However, other factors can make these sensors unreliable. Incorrect mathematical models of the robot can lead to incorrect interpretations of the data. For instance, joint positions are often used to perform forward kinematics, which gives the position of a particular link on the robot given a set of joint angles. If link lengths or joint positions/orientations are incorrect, this leads to error in the final link location. Other unmodeled phenomenon, such as backlash and link deformation, can lead to additional errors.

The combination of these factors can lead to inconsistent reading between sensors. How to fuse these different reading into the most accurate possible interpretation of the robot is still an ongoing research topic. One of the most common ways to combine them is described in the Kalman Filter section.

2.1.2.2 Low Pass filter

A low pass filter is designed to eliminate high frequency noise. It is one of the simplest filters to implement. Low pass filters produce plots like the ones below.

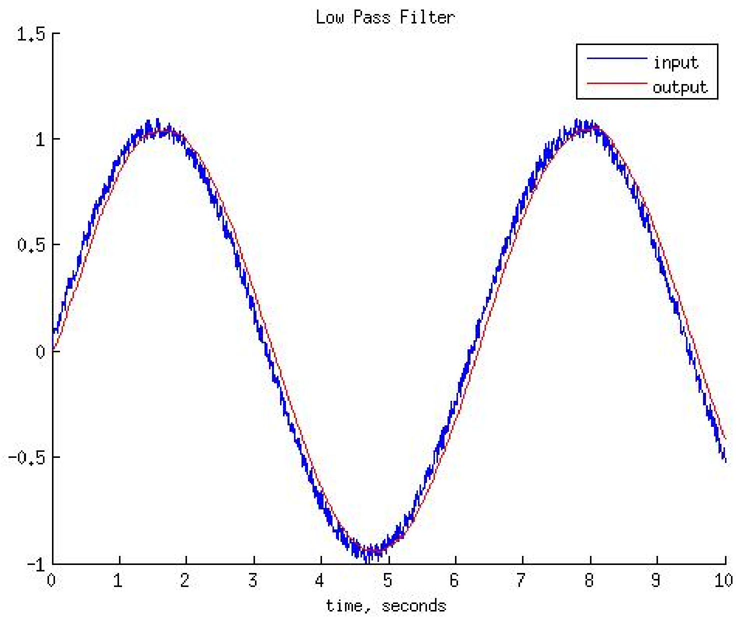


Figure 4: Low pass filter example. The blue line is an incoming noisy signal. The red line is the output after the blue input signal has been filtered. Note how the red line is far smoother than the blue line, but shifted to the right.

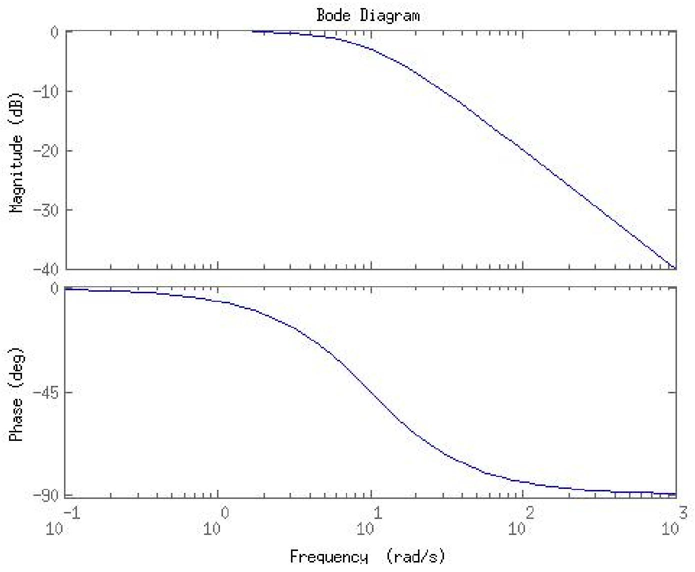


Figure 5: Low pass filter example bode plot. Low pass filters allow signals with low frequencies to pass through, while reducing higher frequency signals magnitude (this is shown on the Magnitude plot). However, low pass filters will shift the phase of the signal, which manifests as a delay in the signal. Depending on the design of the filter, the bode plot will be shifted left or right.

Low pass filters are very easy to implement. A simple example of a low pass filter is shown below:

$$y_k = (1-a)x_k + (a)y_{k-1},\ 0 < a < 1$$

Where *k* is the current time step, *y* is the output of the filter, *x* is the input to the filter, and *a* is a constant parameter. *a* can be modulated to shift the bode plot left or right. This is by no means the only example of a low-pass filter – the term simply applies to filters that eliminate high-frequency noise. These filters are very common in robotics, and are often used to filter noisy sensor data such as joint potentiometers.

2.1.2.3 Kalman Filter

The Kalman filter is a method for fusing sensor data and control inputs from a linear system (Kalman, 1960). Given a control input and perfect modeling for a system, one can predict what state the system will be in next. However, due to modeling errors and other factors, the system rarely matches this prediction perfectly. The Kalman filter takes into account the predicted output and the sensed input and fuses the two into a better estimate. The Kalman filter is based on Gaussians; it keeps track of an estimated state of the system (the mean of the Gaussian) and the "noise" in the system (the covariance of the Gaussian).

Given a linear system denoted by

$$\begin{aligned} x_k &= Ax_{k-1} + B_k u_k \\ y_k &= Cx_k \end{aligned}$$

where k denotes the current time step, x denotes the state of the system, *u* denotes inputs to the system, and *y* denotes outputs from the system, the current state *x* can be estimated by

$$\begin{aligned} \hat{x}_k &= Ax_{k-1} + B_k u_k \\ \hat{P}_k &= A_k P_{k-1} A_k^T + Q \end{aligned}$$

$$\begin{aligned} y_k &= z_k - C\hat{x}_k \\ S_k &= C\hat{P}_k C^T + R \\ K_k &= \hat{P}_k C^T S_k^{-1} \\ x_k &= \hat{x}_k + K_k y_k \\ P_k &= (I - K_k C_k)\hat{P}_k \end{aligned}$$

where *P* denotes the covariance of the estimate (ie the "noise" of the estimate), *Q* denotes the covariance of the noise added by the control input (in this case considered constant), *z* denotes the observed outputs of the system, and *R* denotes the covariance of the noise added by the sensor inputs (in this case considered constant). The top two equations are called the prediction step, while the bottom five are all the update step.

During the prediction step, the filter uses a control input and a model of the system to predict the next state of the system. The filter also adds some amount of noise to the state of the robot, based on the model. In essence, the filter moves the mean of the Gaussian to the next best guess, then increases the magnitude of the covariance.

The update step uses the sensor input to improve the predicted state. The sensor input is assumed to follow a Gaussian distribution, where the mean is the actual state and the covariance models the noise in the sensor. The Gaussian distributions from the prediction step and the sensor input are combined, which reduces the overall covariance.

For a system that is perfectly modeled and is linear, the Kalman filter is actually the optimal algorithm for state estimation. Even for systems that are not perfectly modeled but still linear, the Kalman filter still works very well at estimating the system state. For nonlinear systems, one should use an extended Kalman filter, described below.

2.1.2.4 Extended Kalman Filter

The Extended Kalman filter (EKF) is formulated similarly to the Kalman filter, but is designed for nonlinear systems (Ribeiro, 2004). That is, it can operate on systems defined by

$$\begin{aligned} x_k &= f(x_{k-1}, u_k) \\ y_k &= h(x_k) \end{aligned}$$

where k denotes the current time step, x denotes the state of the system, *u* denotes inputs to the system, and *y* denotes outputs from the system. The EKF is defined by

$$\begin{aligned} \hat{x}_k &= f(x_{k-1}, u_k) \\ \hat{P}_k &= F_k P_{k-1} F_k^T + Q_k \end{aligned}$$

$$\begin{aligned} y_k &= z_k - h(\hat{x}_k) \\ S_k &= H_k \hat{P}_k H_k^T + R_k \\ K_k &= \hat{P}_k H_k^T S_k^{-1} \\ x_k &= \hat{x}_k + K_k y_k \\ P_k &= (I - K_k C_k)\hat{P}_k \end{aligned}$$

where *P* denotes the covariance of the estimate (ie the "noise" of the estimate), *Q* denotes the covariance of the noise added by the control input, *z* denotes the observed outputs of the system, *R* is the covariance of the noise in the sensor inputs, and *F* and *H* are the Jacobians of the *f* and *h* respectively, written as

$$\begin{aligned} F_k &= \left.\frac{\partial f}{\partial x}\right|_{x_k-1,u_k} \\ H_k &= \left.\frac{\partial h}{\partial x}\right|_{\hat{x}_k-1} \end{aligned}$$

While the EKF does enjoy popularity, it is by no means an optimal solution to the estimation problem. For nonlinear systems, further refinements of the Kalman filter exist, such as the Unscented Kalman Filter (Wan & Van Der Merwe, 2000) or the Second Order Kalman Filter (Einicke, 2012).

2.1.3 Control Algorithms

Controls theory refers to the control of inputs to a system in order to produce a specified output. This can range from relatively simple examples, such as using an electric current to drive a motor to a desired angle, to complex systems, such as controlling airfoils on aircraft to a achieve a desired orientation.

Typically, control algorithms operate using a feedback loop to take into account the actual results. This forms the loop shown below.

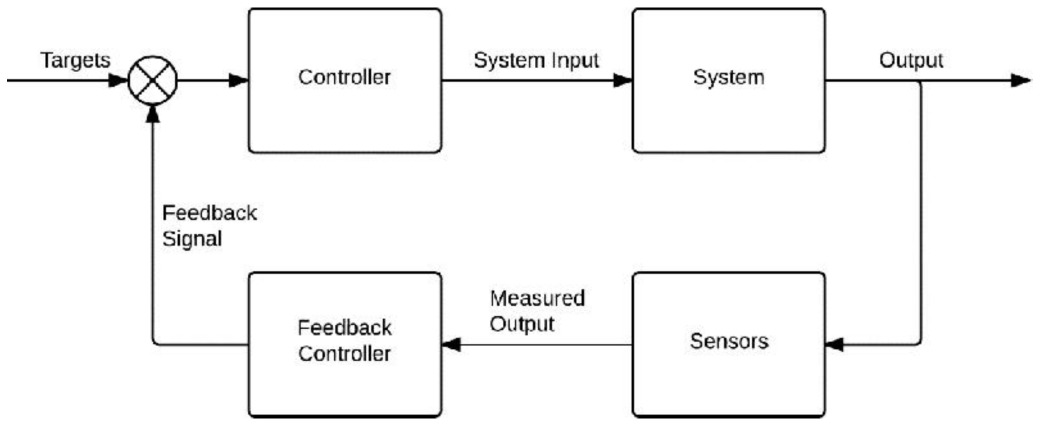


Figure 6: Control system flow chart. The controller is commanded via some targets input, which are sent to the controller. The controller redefines those to be system inputs, which are sent to the system. The sensors read the output of the sensors, which are then interpreted by the feedback controller. The feedback system is then used to refine the target inputs.

While it is often impossible to completely control what the exact output of the system is, there are features of the output that are readily controllable. Loosely defined, these are:

Response time: the amount of time it takes for the output to remain close to the target outputs. Can be thought of as the "speed" of the system, ie how quickly a system can reach a target.

Overshoot: whether a system will reach and then exceed a target, then return to the target.

Oscillations: the tendency of a system to repeatedly overshoot a target.

Steady state error: if the system is left to run for infinite time, any constant remaining error is a steady-state error.

Stability: whether the system will remain near a certain set of states, or whether the output will grow uncontrollably as time goes on.

In general, the ideal response of a control system is to have a quick response time, no overshoot, no oscillations, no steady state error, and to have stability. However, the combination of inaccuracy in actuators, sensors, and system model means that this is impossible. As such, most control systems settle for come compromise between response time, overshoot, oscillations, and steady state error. However, for any control system it is necessary to have stability for any system.

2.1.3.1 PID Control

Proportional-Integral-Derivative (PID) controller is a well-studied control mechanism (Astrom & Hagglund, 1995), generally used for a single variable at a time. The controller is described mathematically as

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt}$$
$$e(t) = x_d(t) - x(t)$$

where $x(t)$ is the state, $x_d(t)$ is the desired state, $e(t)$ is the error between the desired state and the current state, K_p is the proportional gain constant, K_i is the integral gain constant, and K_d is the derivative gain constant. In most implementations, the controller is implemented as a discrete system, and so the integral and derivative functions are replaced by their numerical equivalents (summation and difference).

The proportional K_p term serves to ensure convergence to the desired state x_d. Intuitively, if the error between the current state x and the desired state x_d is large, ie the current state is far from the desired state, then a large input should be used to drive the state to the desired state. As the current state approaches the desired state, smaller inputs are required to drive it the rest of the way.

The integral term K_i is used to reduce the steady-state error to zero. In many scenarios, there are external forces that will prevent the system from reaching its target position using only a proportional gain. That is, at some point the input *u(t)* generated by $K_p e(t)$ will be negated completely by an external force, even if *e(t)* is not zero. This results in zero motion, so the error will remain at a single value. This is known as the steady-state error. It is often caused by external forces such as gravity or static friction. By having an integral term, the constant error will be added over time, and so *u(t)* will increase in magnitude over time, eventually overcoming the external forces.

The derivative term K_d is used to prevent overshoots and oscillations. If K_d is positive, as the error decreases the derivative term generally opposes the proportional term, reducing the overall magnitude of *u(t)*. While this may seem counterproductive, having only a proportional term will often cause overshoots and oscillations. Having a derivative term serves to damp the oscillations and prevent overshoot.

In general, increasing the proportional gain will improve system response time (speed) and reduce steady-state error but can cause oscillations and overshoot, increasing the integral term will eliminate steady-state error faster but will cause oscillations and overshoot, and increasing the derivative term will reduce the system response time but reduces/eliminates oscillations and overshoots.

PID does not allow a user to completely control the output features discussed above. In fact, it does not guarantee stability, and so a system's output can grow uncontrollably even under the control of PID. There are variations on PID that allow these features to be more controllable.

2.1.3.2 PDFF Control

Proportional-Derivative-Feed-Forward (PDFF) controllers were developed to combat steady-state error. The mathematical The Proportional and Derivative terms remain the same as in PID. The feed-forward term is based on a model of the system, and accounts for the input that does not depend on the error in the system. The controller is designed as follows:

$$u(t) = K_p e(t) + K_d \frac{de(t)}{dt} + FF(x(t))$$

As an example, the feedforward component can be used for gravity compensation. Given a joint that needs to lift a weight, the feedforward term calculates the input required to oppose gravity. That is, given *e(t)=0*, the feedforward term *FF(x(t))* calculates the input is needed to keep the joint stationary. In this way, the feedforward component will eliminate any steady-state error, assuming that it can perfectly compensate for external forces.

Since the feedforward component serves to eliminate steady-state error, this makes it partially redundant with the integral term. Removing the integral term can often reduce system overshoot and oscillation, and in general makes the system more stable.

2.1.3.3 Combinations of PID(FF) Controllers

For most systems, there can be multiple target states but only one input. A simple example is a one-joint electric robot arm. The only input to the system is the voltage applied to the joint, but the outputs include the position, velocity, acceleration, and torque of the joint. A PID(FF) controller can be applied to each output term, and the input to the system is the sum of all the controllers. This can be advantageous in some situations, such as trajectory tracking.

2.1.4 Motion Planning

While control normally acts on the current time step, motion planning determines the actions that should be taken over a large span of time in the future. Motion planning is another well-researched area, with new algorithms appearing regularly.

Typically, a motion planner creates a trajectory for the robot to follow. In theory, these trajectories should be continuous time functions that move smoothly between points in space. In practice, trajectories are often divided into discrete waypoints that are operated on individually. When using waypoints, the robot will interpolate between each waypoint.

The motion planning problem is to find a path for the robot to follow that satisfies some constraints. The simplest problem is moving from one configuration to another – that is, moving from a starting state to a goal state. In this sense, only two constraints are applied; the first waypoint of the trajectory must be the starting state, and the final waypoint must be the end state. The next commonly seen constraint is collision avoidance; that is, for all waypoints and paths between the waypoints. Other constraints can also be added, such as balance constraints or kinematic chain constraints.

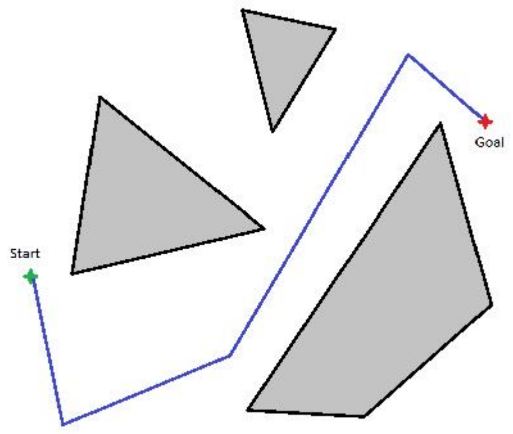


Figure 7: A simple motion planning problem. The only constraints are the initial start point, the goal point, and collision avoidance. The grey polygons are obstacles. The green point is the start point, and the red point is the goal point. The blue path is a solution to the motion planning problem.

Motion planning is almost universally performed in configuration space, as opposed to task space. As explained before, configuration space is the space of all possible robot configurations or states, and for the sake of motion planning is normally considered to be the space of all possible joint values. In other words, if there are 10 joints on the robot, then there are 10 unique variables that correspond to each joint on the robot. This means the configuration space is R^{10}. The task space is where real world objects are defined, and is analogous to the physical world. Obstacles are defined in task space, and many goals are also defined in task space as well. The robot also exists in task space – the shape and location of the robot is dependent on the position of the robot in configuration space.

There are a number of motion planning algorithms commonly used today, such as Probabilistic Road Maps (PRM) (Geraerts & Overmars, 2004), Rapidly-exploring Random Trees (RRT) (LaValle & Kuffner, 2000), and various Optimization methods such as CHOMP (Ratliff, Zucker, Bagnell, & Srinivasa, 2009, May) and Trajopt (Schulman, et al., 2013, June). PRMs and RRTs are sampling-based methods that use random points to explore a space. PRM place waypoints into the configuration space and then connecting to form a roadmap. Once the roadmap is created, the algorithm simply executes a graph search to find a path from goal to finish. RRTs generate random trees in the configuration space until the start is connected to the goal. Optimization methods refine an input trajectory until it is "optimal", although in practice this optimal point is a local optimum rather than a global optimum.

Sampling based methods are probabilistically complete, meaning that if left to run for infinite time they will find a path from the start to goal, if such a path exists. RRT and PRM in their most basic form are not optimal; the paths they return will not be the shortest, fastest, safest, etc, and will instead look somewhat erratic. This is improved by the use of a smoothing function, but this still will not give optimality. Recent variants of PRMs and RRTs have given asymptotic optimality, meaning that if the algorithm runs forever, the path returned will be optimal. These work by continually improving the paths returned, and so if the algorithm is stopped early, it will not have an optimal path.

Optimization methods are not complete in any sense: for a given input trajectory, there is no guarantee a feasible trajectory will be generated, rather than a trajectory that violates some constraints such as collision avoidance. This is because optimization methods are roughly equivalent to gradient descent, which can fall into local minima. These local minima are not guaranteed to satisfy the constraints.

2.2 Atlas

We performed the work in this paper on the Boston Dynamics Atlas. All of the work on Atlas was for the DARPA Robotics Challenge (DRC). We were provided with a robot, and as such we developed very little in the way of hardware, focusing almost purely on software and algorithms.

This section will first describe the DRC in order to provide the context for our work. Then we will explain the actual hardware and the tools provided by Boston Dynamics and DARPA. Finally, we will describe the algorithms and methods implemented by our team on the robot, which will provide a backbone for the work done by the author.

Much of this data can also be found in (DeDonato, et al., 2015) and (Atkeson, et al., 2015).

2.2.1 DRC

The DARPA Robotics Challenge (DRC) was a competition where robots attempted to complete a series of tasks modeled after a disaster zone (Pratt & Manzo, 2013). It was originally conceived after the Fukushima nuclear disaster in 2011, where robots were equipped to observe the damage but were unable to act upon the information. As such, the DRC includes tasks such as operating valves and navigating through rubble or difficult terrain.

The DRC consisted of four different categories for teams: A track teams were given funding to develop their own robotic platforms for the challenge; B track teams were given funding to develop software for controlling a Boston Dynamics Atlas; C track teams were unfunded, but could compete for funding and an

Atlas; and D track teams were unfunded and were not given any hardware. This paper covers research done by the WPI-CMU team, which was a C track team.

The DRC consisted of three phases, described below.

2.2.1.1 Virtual Robotics Challenge

Before being given an Atlas, both B and C track teams needed to compete in the Virtual Robotics Challenge (VRC). The VRC was done in simulation. Teams were expected to complete three different simulated courses, which included ingress, egress and operation of a vehicle, navigating difficult terrain, and attaching a hose to a port. Following the VRC, winning teams were awarded the Atlas as well as funding.

2.2.1.2 DARPA Robotics Challenge, Trials

The next phase of the competition was the DRC Trials. The trials were outdoors at Florida's Homestead Miami Speedway, with all teams needing to bring their robots on site. The Trials consisted of 8 different events, completed in separate 30 minute events. They were:

- Vehicle task: drive a vehicle through an obstacle course and egress
- Terrain task: navigated a collection of uneven cinderblocks and other obstacles
- Ladder task: climb a naval ladder
- Debris task: clear a narrow area of debris and pass through
- Door task: open a series of doors with increasing difficulty
- Wall task: drill a hole in a sheet of sheetrock
- Valve task: operate a series of valves
- Hose task: retrieve a hose and attach it to a wall hose thing (?)

For each task, a maximum of 4 points could be scored, giving a total of 32 points possible.

A, B, and C track teams were required to compete, while participation by D track teams was optional. The top 8 teams received continued funding, although due to later events the top 9 teams received funding.

2.2.1.3 DARPA Robotics Challenge, Finals

The last phase of the competition was the DRC Finals. The finals were held outdoors in Pomona, California. The competition consisted of 8 different tasks to be completed within one 60 minute run. The tasks were:

- Vehicle task: drive a vehicle through an obstacle course
- Egress task: exit the vehicle
- Door task: open and walk through a door
- Valve task: turn a valve
- Wall task: drill a hole in sheetrock
- Surprise task: a task announced at the Finals; either pulling a lever or pushing a button
- Rubble task: either walking through light debris or walking over rough terrain
- Stair task: climb a set of stairs

Each task was worth one point. Ties were broken with time, with faster times being better. The top three teams won prize money.

Unless otherwise specified, this report will refer to the DRC Finals and methods developed during that phase of the challenge.

2.2.2 Systems

The Boston Dynamics Atlas v5 platform is a 30 DoF humanoid robot. It is a hydraulic electric hybrid system, equipped with an IMU, a spinning lidar, stereo vision cameras, and force-torque sensors on the hands and feet. Of the 30 DoF, there are 6 in each leg, 3 in the torso, 1 in the neck, and 7 in each arm. The neck joint is electric, as are the last 3 joints in each arm. It is able to equip a battery pack which gives it upwards of an hour of untethered time.

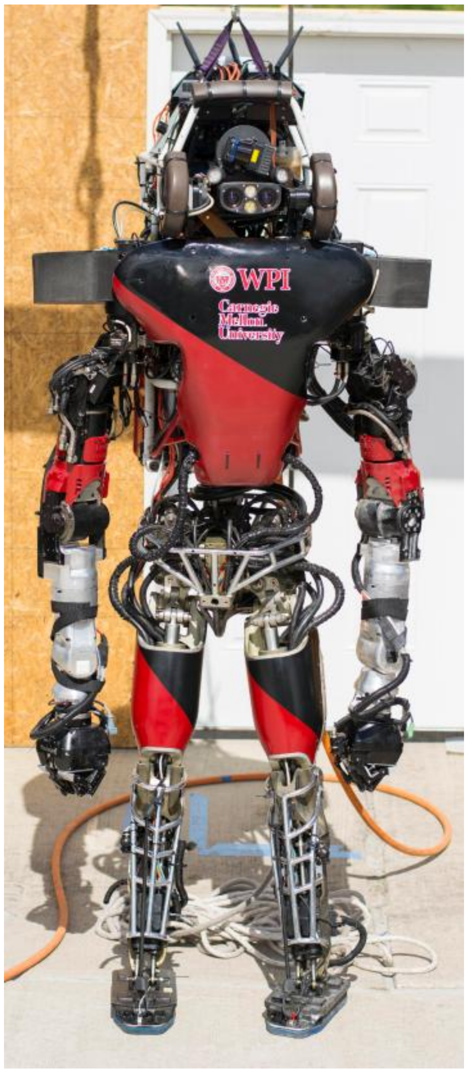


Figure 8: WPI's Boston Dynamics Atlas.

Also of interest is the Boston Dynamics Atlas v3 platform, which is a 28 DoF humanoid robot. Unlike its successor, it is fully hydraulic and 6 DoF arms. It cannot mount a battery, and thus must be tethered. Additionally, the shoulders are mounted at a different angle than on Atlas v5. This was changed to allow for a better manipulation workspace. This version of Atlas was used for the DRC Trials.

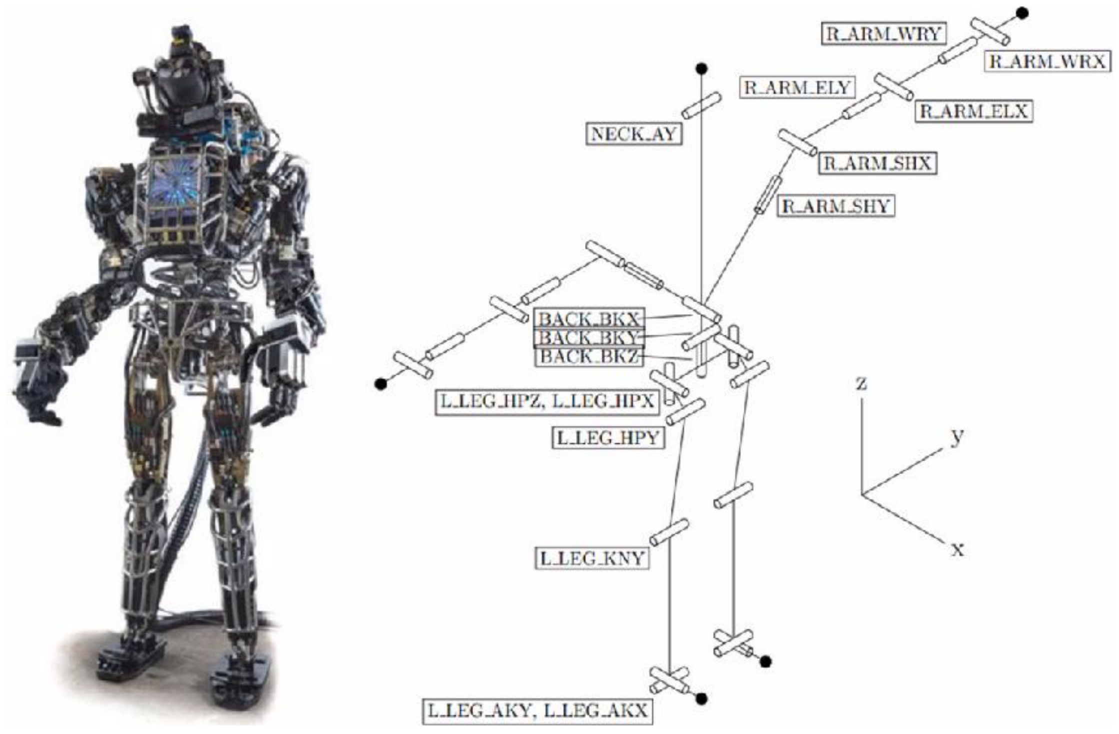


Figure 9: The original Atlas v3 platform, and corresponding joint diagram. This is the initial design provided to DRC teams in 2013.

We had a selection of hands to choose from. Originally provided were the Sandia hand and the iRobot hand. Both hands have options for tactile sensing. The Sandia hand also contains 2 cameras for stereo vision. However, we found that both of these lacked the strength and robustness we desired. In the end, we chose to use the Robotiq 3-Finger Adaptive gripper. While it does not contain as many sensors as the Sandia hand and the iRobot hand and is arguably not as dexterous, the extra robustness and ease of use was more than enough for use to choose it. Also provided for testing was the SRI hand, which provided additional strength and robustness at the cost of dexterity. However, due to its late introduction and some other flaws, we chose to remain with the Robotiq hands.

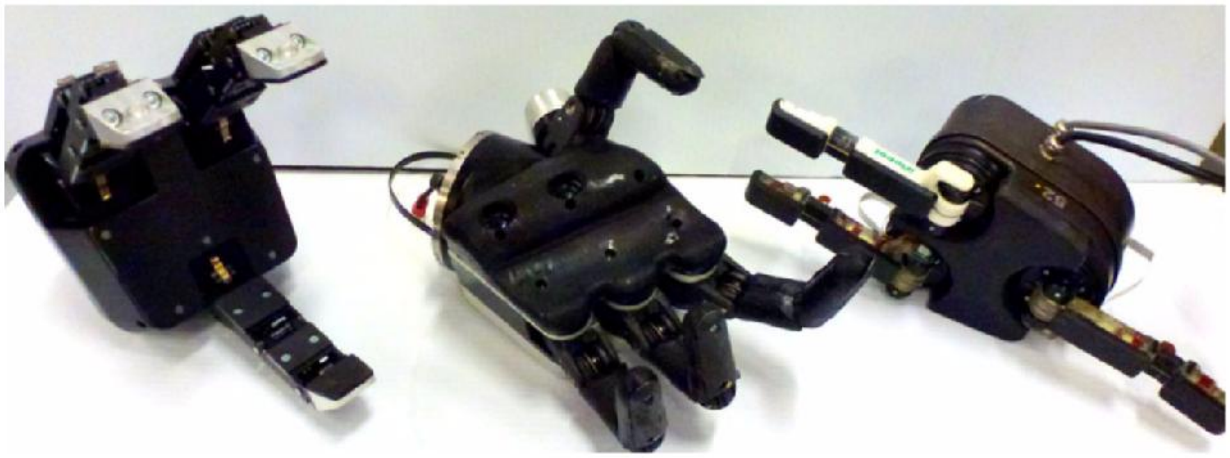


Figure 10: The three different hands the WPI-CMU DRC team tested. From left to right: Robotic 3 Finger Adaptive Gripper; Sandia hand; iRobot hand. The Robotiq hand was eventually selected for use.

Our Atlas also had additional sensors mounted. We mounted cameras on the hands and the ankles. This provided extra vision in areas difficult for the head to see. These extra cameras proved vital in the competition. For research, additional IMUs were attached to the robot at various points to allow for better state estimation.

The system architecture consisted of 4 onboard computers. One of these computers was used by BD, and was unavailable to our team for use. However, the other three computers were open to us. We used the Robot Operating System (ROS), which provides inter-process communication. The ROS system also provides many developed libraries and packages for use.

2.2.3 Algorithms

This section details the specific implementations of various algorithms used on the Atlas platform.

2.2.3.1 State Estimation

We used a series of low-pass filters for the incoming joint values. This reduced to overall noise from the sensors to a usable amount. We also used low-pass filters for the joint velocities and joint accelerations, which were calculated based on the difference in joint states between timesteps.

We used a steady-state Extended Kalman Filter to estimate the pelvis position and orientation in the world frame. As time approaches infinity, the prediction error variance for a kalman filter will approach a single value if the state is constant. As such, the steady state kalman filter uses this variance in its calculation, removing much of the complexity in the algorithm and guaranteeing numerical stability.

The Kalman filter acts on the pelvis pose calculated by the kinematics of the legs and the pelvis pose given by the IMU. For orientation, the Kalman filter is heavily weighted to use the IMU orientation estimate, while for the position the filter is weighted to use the kinematics almost exclusively. This does assume that at least one foot is on the ground at all times.

There were also a number of other state estimation techniques that were tested on the Atlas platform. In some cases, these were used alongside our manipulation methods. However, for the DRC Finals we only used the steady-state EKF. The addition state estimation methods are described in (Xinjilefu, 2015).

2.2.3.2 Control

There are two major layers of control on the robot.

The lowest level are controllers on each joint. These controllers are designed to allow control of position, velocity, and torque, and are structured as PID or variants thereof.

The second layer has two options for control. One is the BDI supplied control schemes, and the other was developed in house.

The BDI control method comes in several different modes – stand, step, walk, and manipulate. Stand is a static balancing controller, and does not allow for any other motions other than those needed to balance. Step and walk for locomotion; step allows for precise foot placement, while walk is a quick dynamic set of steps and does not allow for footstep placement. Finally, manipulate gives the user control over the pelvis height, torso joints and arm joints, and uses the legs to maintain balance during arm motions. This means that the user has limited control over the end effector position, since the final state of the leg joints is determined by the controller and not the user.

Although the controller is supposed to prevent falls, in practice we found that it was quite easy to cause them. For manipulation, this normally occurred when the robot was at the edge of its workspace or when the robot was in contact with its environment. Nonetheless, the BDI supplied controllers were sufficient for performing most of the tasks if caution was used.

The other method of control uses online hierarchical optimization to calculate joint torques in real time (Feng, Whitman, Xinjilefu, & Atkeson, 2015). This controls the entire body at once, allowing for synchronized movements that allow it to maintain the robot's balance during any motion. It uses quadratic programming to solve for inverse dynamics and inverse kinematics at each time step. We found that it was more reliable than the supplied BDI behaviors, and so during the DRC finals we used it exclusively.

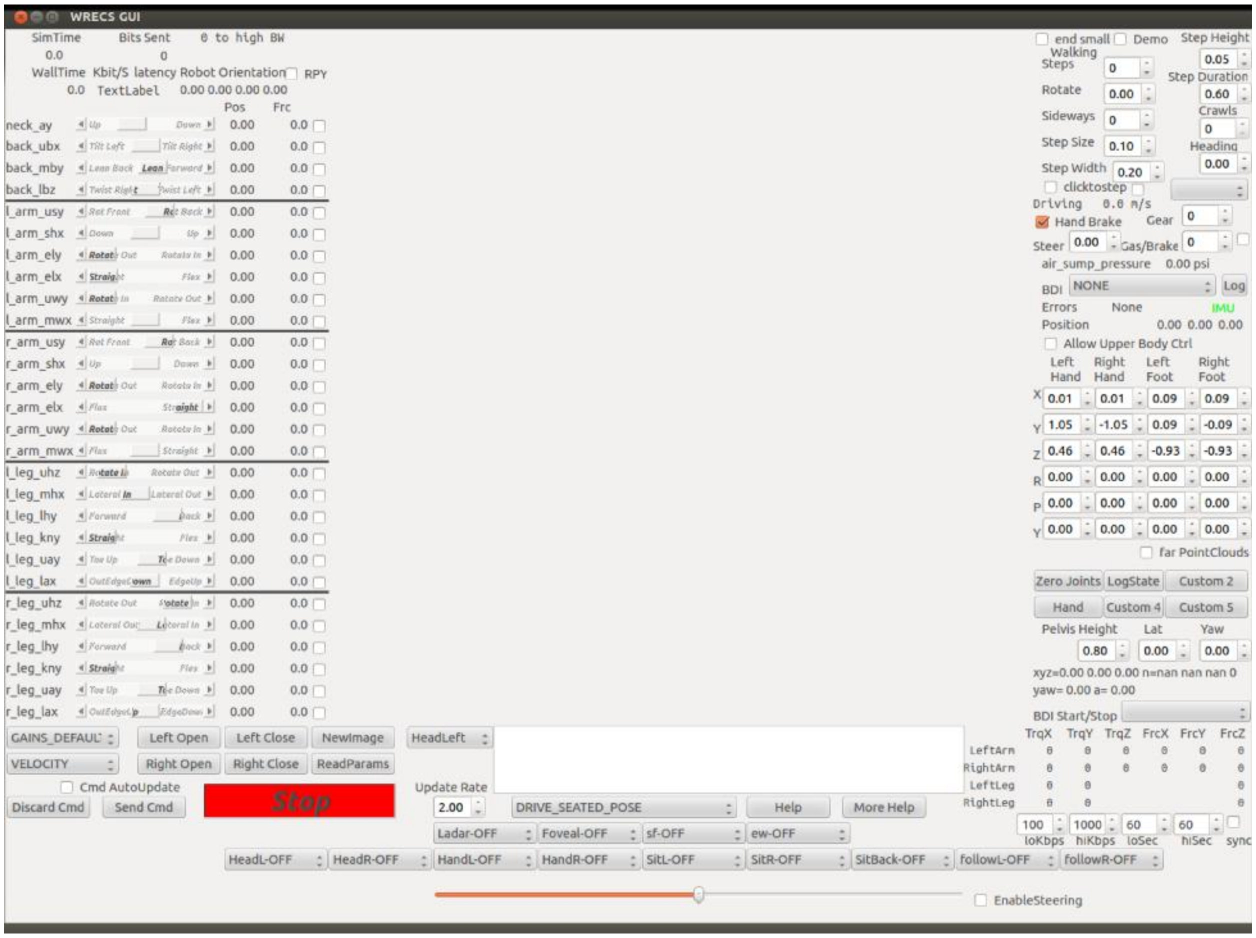
3 Interface for Manipulation

Over the course of the DRC, the graphical user interface (GUI) was designed and redesigned multiple times. Since we were constantly upgrading and modifying the capabilities of Atlas, this was to be expected. We also experimented with teleoperation techniques other than simple keyboard and mouse, such as using a Razer Hydra for manipulation. However, this report will only cover the methods developed at least in part by the author.

For the Sarcos platform, online control of the robot was restricted to a command line interface that comes with the SL control system. As such, it will not be covered in this report.

3.1 Original Design

The first design for the manipulation GUI was embedded in Rviz as a supplement to the primary interface, WGUI (shown below). WGUI provided access to many of the basic functions needed on Atlas, such as changing control modes, activating the pump, commanding walk and steps, and some limited IK. However, we found that we needed more tools in order to effectively control the robot for manipulation.



SimTime 0.0 Bits Sent 0 0 to high BW

WallTime 0.0 Kbit/S latency TextLabel Robot Orientation 0.00 0.00 0.00 0.00 RPY

	Pos	Frc
neck_ay	0.00	0.0
back_ubx	0.00	0.0
back_mby	0.00	0.0
back_lbz	0.00	0.0
l_arm_usy	0.00	0.0
l_arm_shx	0.00	0.0
l_arm_ely	0.00	0.0
l_arm_elx	0.00	0.0
l_arm_uwy	0.00	0.0
l_arm_mwx	0.00	0.0
r_arm_usy	0.00	0.0
r_arm_shx	0.00	0.0
r_arm_ely	0.00	0.0
r_arm_elx	0.00	0.0
r_arm_uwy	0.00	0.0
r_arm_mwx	0.00	0.0
l_leg_uhz	0.00	0.0
l_leg_mhx	0.00	0.0
l_leg_lhy	0.00	0.0
l_leg_kny	0.00	0.0
l_leg_uay	0.00	0.0
l_leg_lax	0.00	0.0
r_leg_uhz	0.00	0.0
r_leg_mhx	0.00	0.0
r_leg_lhy	0.00	0.0
r_leg_kny	0.00	0.0
r_leg_uay	0.00	0.0
r_leg_lax	0.00	0.0

GAINS_DEFAUL, VELOCITY, Cmd AutoUpdate, HeadLeft, Update Rate 2.00, DRIVE_SEATED_POSE

Ladar-OFF, Foveal-OFF, sf-OFF, ew-OFF

HeadL-OFF, HeadR-OFF, HandL-OFF, HandR-OFF, SitL-OFF, SitR-OFF, SitBack-OFF, followL-OFF, followR-OFF

EnableSteering

end small, Demo, Step Height 0.05

Walking Steps 0, Step Duration 0.60

Rotate 0.00, Crawls 0

Sideways 0, Heading 0.00

Step Size 0.10

Step Width 0.20

clicktostep

Driving 0.0 m/s

[✓ Hand Brake] Gear 0

Steer 0.00 Gas/Brake 0

air_sump_pressure 0.00 psi

BDI NONE

Errors None IMU

Position 0.00 0.00 0.00

Allow Upper Body Ctrl

	Left Hand	Right Hand	Left Foot	Right Foot
X	0.01	0.01	0.09	0.09
Y	1.05	-1.05	0.09	-0.09
Z	0.46	0.46	-0.93	-0.93
R	0.00	0.00	0.00	0.00
P	0.00	0.00	0.00	0.00
Y	0.00	0.00	0.00	0.00

far PointClouds

Pelvis Height 0.80, Lat 0.00, Yaw 0.00

xyz=0.00 0.00 0.00 n=nan nan nan 0

yaw= 0.00 a= 0.00

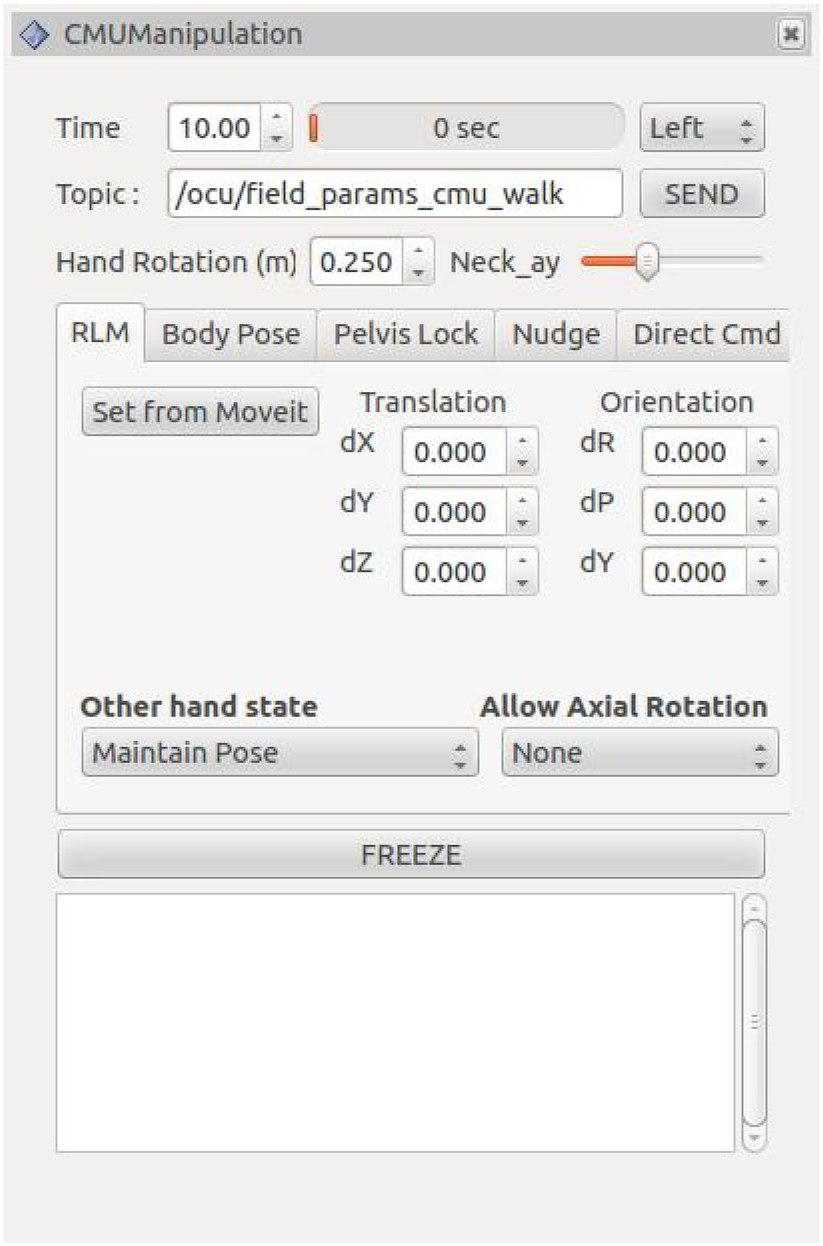
BDI Start/Stop

	TrqX	TrqY	TrqZ	FrcX	FrcY	FrcZ
LeftArm	0	0	0	0	0	0
RightArm	0	0	0	0	0	0
LeftLeg	0	0				0
RightLeg	0	0				0

loKbps	hiKbps	loSec	hiSec	sync
100	1000	60	60	

Figure 11: The original WGUI used by the WPI-CMU DRC Team. On the left are sliders that control each of the 28 joints. Along the bottom are various options for sensors and systems. On the right are controls for walking, forward kinematics and inverse kinematics, and other sensor data.

Rviz is a visualization tool integrated with ROS. It allows users to visualize various sensors and provides an interface for issuing commands. The original manipulation GUI (CMUManipulation) was designed as a plugin for Rviz, and allows users to send commands specific to Atlas and our software.



Time 10.00 0 sec Left

Topic: /ocu/field_params_cmu_walk

Hand Rotation (m) 0.250 Neck_ay

dX 0.000 dR 0.000

dY 0.000 dP 0.000

dZ 0.000 dY 0.000

Other hand state: Maintain Pose

Allow Axial Rotation: None

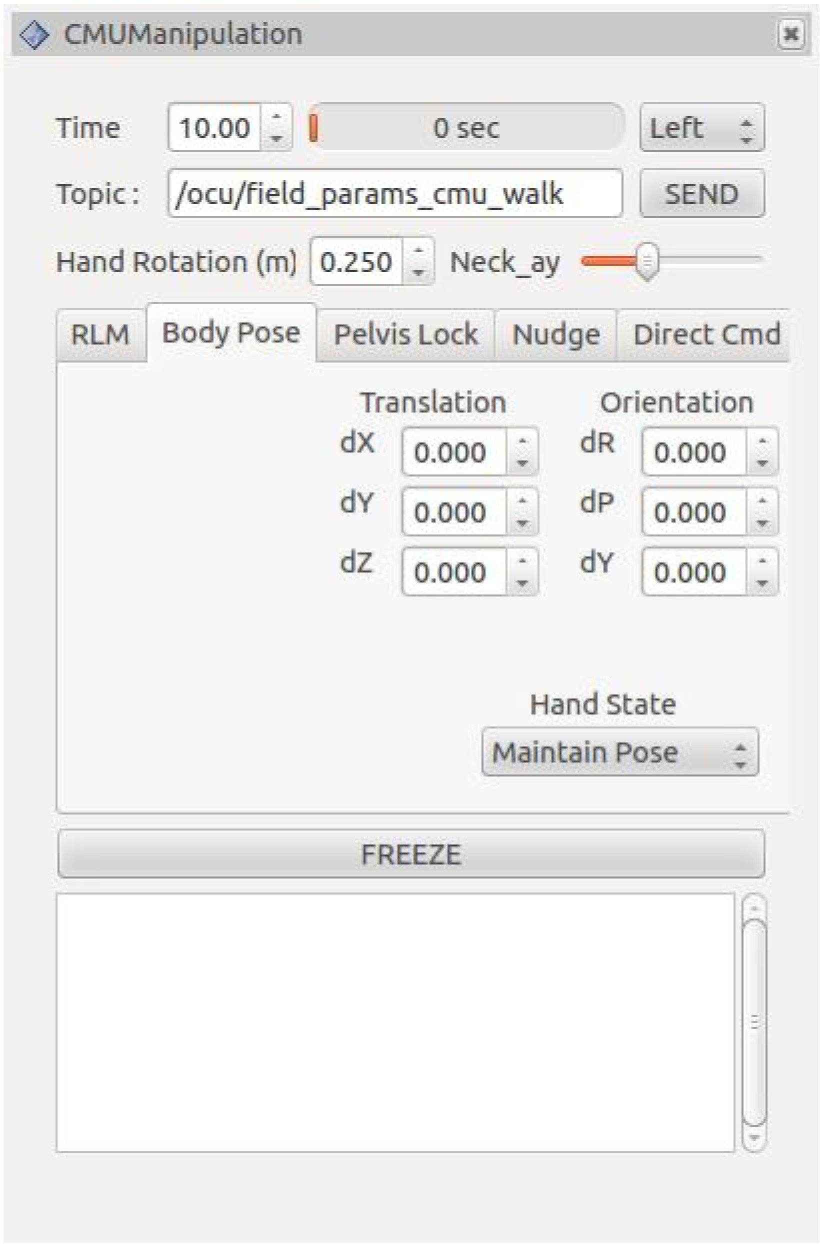
Figure 12: CMUManipulation tool, Relative Linear Motion tab. In this tab, there are options for generating motions that track a straight line in task space.

The window contains general parameters for manipulation. These are the time for execution, which side of the robot is used (left vs right), what topic the commands are published to, a controller for using the neck joint. There is also parameter for hand rotation, which sets the location the end point for manipulation. CREATE IMAGE TO EXPLAIN. There are also two command buttons, one for the sending the command to the robots, and one for freezing the robot in case of emergency. The text box on the bottom listens to a feedback topic and allows the robot to report status and parameter settings.

There are also 5 tabs for setting different commands.

The first tab, labeled RLM, sends relative linear motion commands. This command takes a desired change in position and orientation, and creates a straight line trajectory to final location. That is, the end effector will follow a straight line to the final location. We used this for larger motions where the position of the end effector needed to be predictable. For instance, when using a power tool such as a drill, we needed to be sure what path the drill bit would take.

The user can set the desired changed in location (ie Translation) and the desired change in Orientation. In addition, the user can specify what the other hand should do during the motion (Other hand state). The options are Maintain Pose, which sets the other hand to maintain its position and orientation in relation to the world frame, or Lock Joints, which set the other arm to maintain its joint positions. In general, Maintain Pose can be thought of as a constraint that affects the whole body, and is only used when a task requires it (ie when both arms are interacting with the world). As such, we mainly used Lock Joints, as this does not place any constraints on the rest of the body.


CMUManipulation

Time 10.00 | 0 sec | Left

Topic : /ocu/field_params_cmu_walk | SEND

Hand Rotation (m) 0.250 | Neck_ay

RLM | Body Pose | Pelvis Lock | Nudge | Direct Cmd

Translation		Orientation	
dX	0.000	dR	0.000
dY	0.000	dP	0.000
dZ	0.000	dY	0.000

Hand State: Maintain Pose

FREEZE

Figure 13: CMUManipulation tool, Body Pose tab. This provides options for shifting the torso and CoM position.

The next tab sets torso motions, roughly speaking. The user can specify a change in CoM, which will be achieved primarily using the legs. The orientation applies for both the torso and the pelvis together, ie changes in orientation will apply to the orientation of both the torso and the pelvis.

This control method was added to give us a larger workspace. Since the robot's CoM is normally fixed, this can severely limit how for the robot can reach as leaning is not an option. However, in practice we mostly left the CoM orientation, X location and Y location fixed and did not alter it. Moving the CoM in those directions significantly increases the chances of following.

The option for hand state sets the targets for each hands during the motion. The options are Maintain Pose, which sets the hands to maintain their location and orientation in the world frame, and Lock Joints, which locks the arm joint angles (maintains their location and orientation in the torso frame).

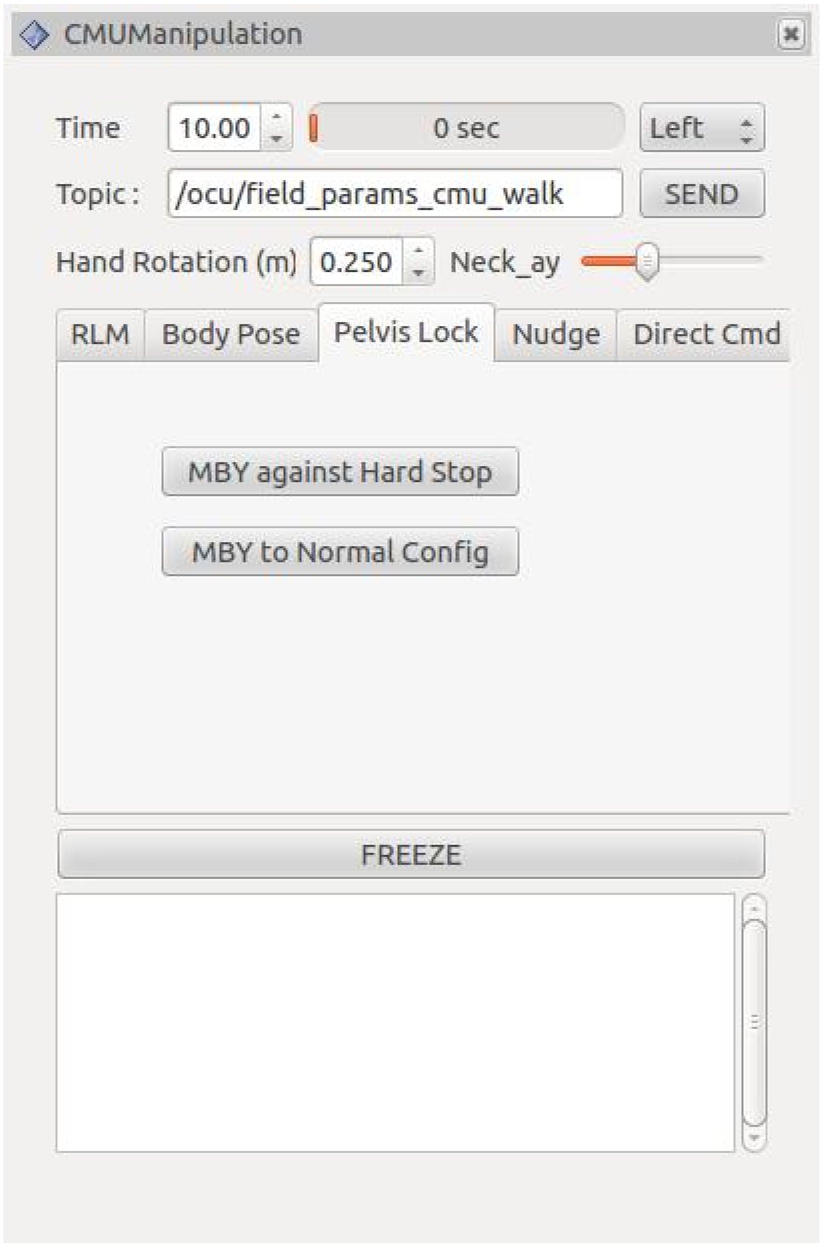
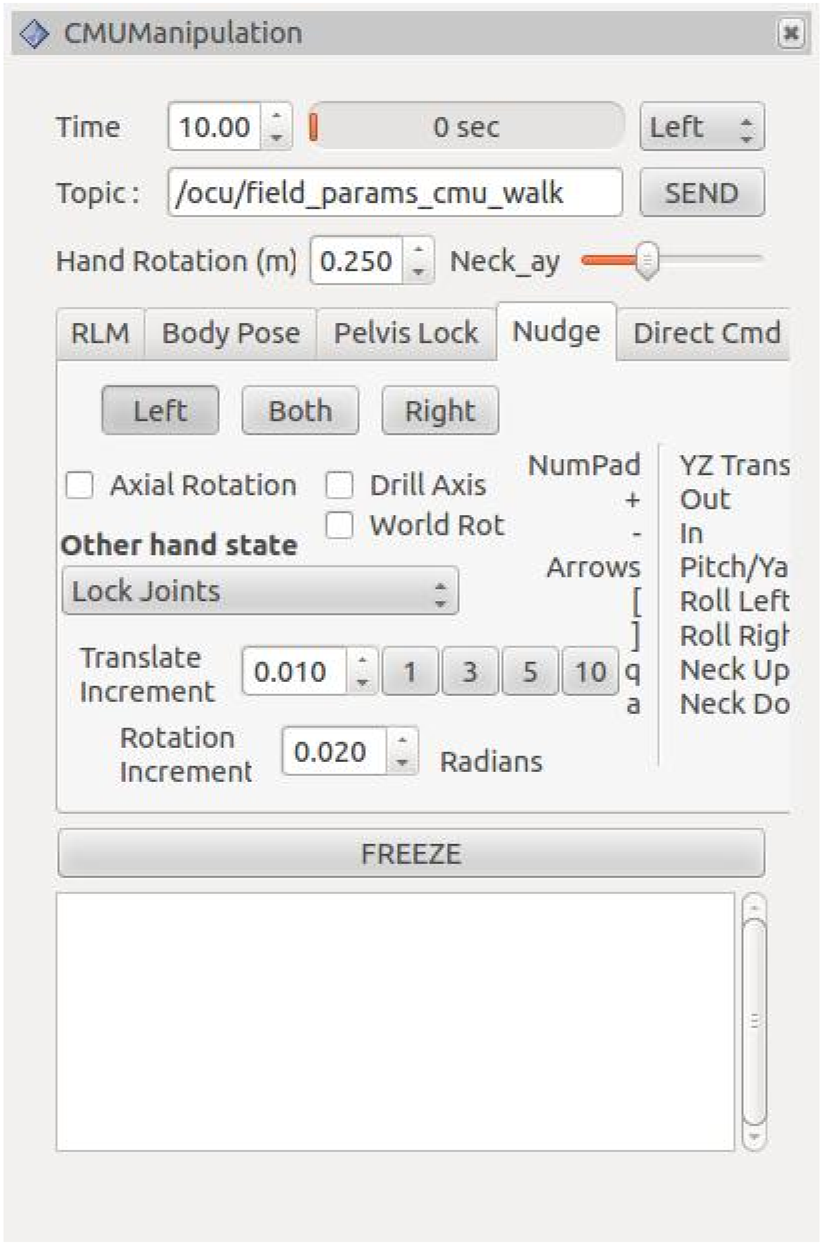


Figure 14: CMUManipulation tool, Pelvis Lock tab. This tab gives two buttons that push the torso Y joint (which controls leaning forwards and backwards) to its mechanical end stop.

The Pelvis Lock tab allows users to execute a specific motion as a temporary solution to a design issue with Atlas. The pelvis y joint (the joint that leans the robot forward and backwards) was not strong enough in the first version of Atlas we received, and so could not always apply enough torque to prevent the robot from leaning to the hard stop. Normally, the back joint is close straight up and down, and so the CoM of the upper body is centered above the pelvis y joint. This means the torque applied by gravity to the joint is low. Once the robot begins leaning forward, the torque significantly increases. If the torque due to gravity exceeds the maximum torque the pelvis joint can apply, then the robot will lean forward until it reaches the hard stop for that joint. This is very undesirable, as the robot will have an uncontrollable descent to the hard stop, followed by a sudden jerk caused by the hard stop.

The Pelvis Lock command pushes the pelvis y joint to the hard stop in a safe and controllable manner. It does this by rotating the pelvis instead of leaning the robot forwards. This allows the upper body CoM to stay above the pelvis y joint, so the joint's torque is maxed out. Once the pelvis y joint is pushed to the hard stop, the robot can lean forward further. This is because the hard stop can apply effectively infinite torque. As such, the workspace of the robot is increased.



Time 10.00

Topic : /ocu/field_params_cmu_walk

Hand Rotation (m) 0.250

Other hand state Lock Joints

Translate Increment 0.010

Rotation Increment 0.020 Radians

NumPad	YZ Trans
+	Out
-	In
Arrows	Pitch/Ya
[	Roll Left
]	Roll Rigl
q	Neck Up
a	Neck Do

Figure 15: CMUManipulation tool, nudging tab. This tab provides setting for "nudging", a form of teleoperation used heavily by the WPI-CMU DRC team. Commands for nudging are input via keyboard.

The Nudge tab allows users to control the end effector(s) using small tele-operated motions. The parameters allow the user to control how the motions are performed. The user can choose between controlling the left hand, the right hand, or both at the same time. The options for rotation (Axial Rotation, Drill Axis) control whether the motion will maintain the end effector's orientation or not. This was used for the drill task, where the drill could rotate around its major axis and still cut effectively. By releasing the orientation constraint for the end effector, the robot can move much more freely. The world rotation option chooses whether the nudges are performed in relation to the world frame or in the hand frame. The Other hand state controls what the other hand does during nudging, and does nothing if the user is controlling both hands. The options are Maintain Pose, which sets the hands to maintains their location and orientation in the world frame, and Lock Joints, which locks the arm joint angles (maintains their location and orientation in the torso frame). The translate increment sets how far the end effector will move for each translational nudge. The rotation increment sets how much the end effector will rotate for each rotational nudge.

The actual nudging is performed using the keyboard (the keybindings are shown on the panel). Typically, the operator will watch either the virtual visualization of the robot or the incoming camera data, and use nudging as a way to teleoperate the robot. For motions that needed to be precise and accurate, this is the best method we had.

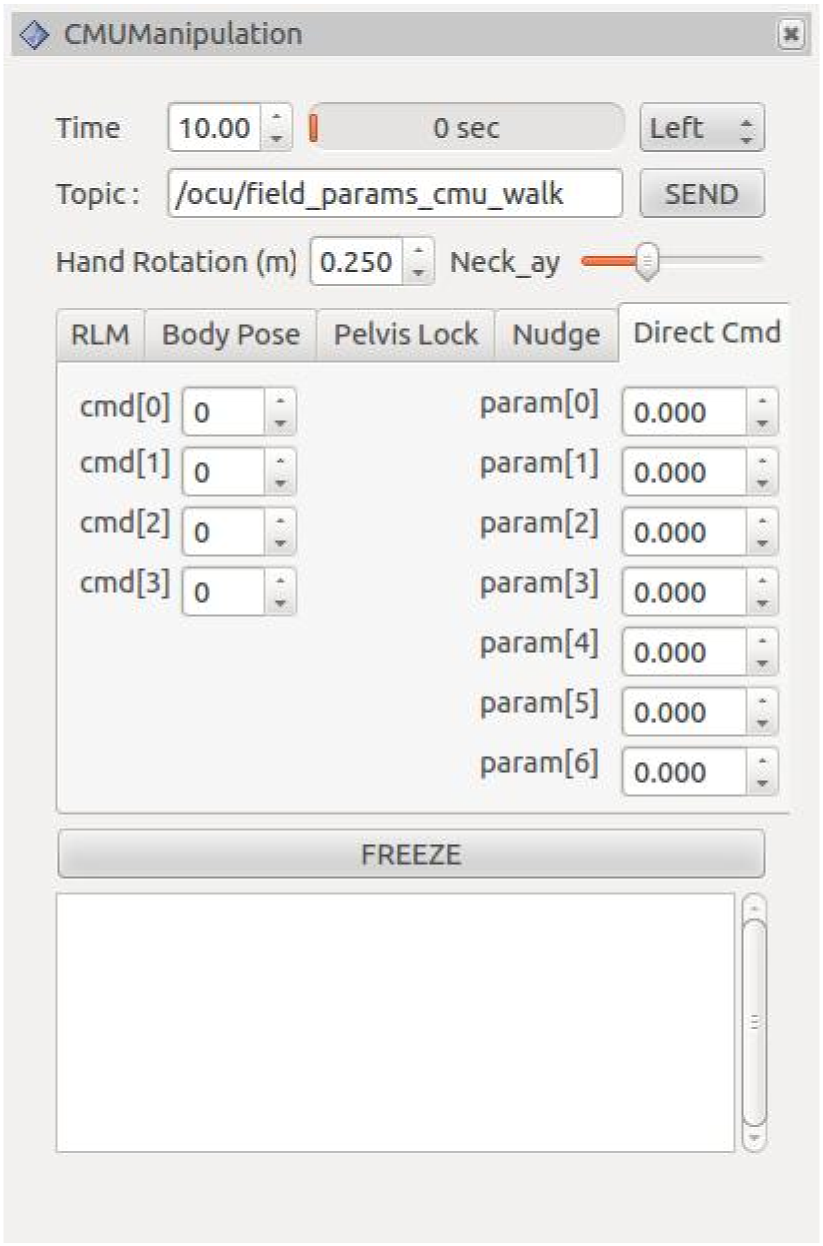


Figure 16: CMUManipulation tool, Direct Command tab. This tab allows users to send commands as they are coded in software.

The Direct Cmd tab allows users to send commands directly to controller. The other tabs parse the human readable settings and place them into a field_param message, which is formed as a set of commands and parameters. Each command has a specific set of parameters that is defined in an API. The API in question can be found in Appendix A.

3.2 Lessons Learned

Although this interface was used for some tasks, many members of the team opted not to use it. Part of this was merely a lack of communication between different sub-teams and so some members of the team were unaware the panel existed, but much of the reason was that the panel did not fulfill their needs. There were also some design flaws in the panel. The main drawbacks are as follows:

Excessive design: The panel allowed far more control that was actually necessary. The most blatant example of this is the direct command tab. Although this could be useful for debugging, as it allowed full control over the parameters and commands sent, it was unwieldly for the average operator. There were also generally more options than were required. At the DRC Trials, the most used tab by far was the nudging tab. The other tabs were used far less. For motions that could be created using those tabs,

Lack of preplanned motions: For many tasks, it was easy to predict what motions would be needed. For instance, the valves were placed at a known height with a known orientation. As such, the first step for the valve task could be a pre-planned motion that lifted the hand to that height and orientation. This makes further motions much easier to design. The original GUI designs did not account for this. In response, many members of the team created their own GUIs, consisting of buttons that would generate these pre-planned motions. This lack of centralization was possible in the DRC Trials because of the organization of the competition. Each task was performed separately, and so each task could have its own startup sequence which would include the custom GUI. For the DRC Finals, however, the tasks needed to be performed sequentially. Therefore, we needed to centralize all of the GUIs into a single program.

Lack of visualization: Although it was possible to generate a number of different motions, there was no utility for viewing what the motion would look like in advance. In particular, when using the GUI it could be hard to tell what the final position of a motion would be. For instance, the Relative Linear Motion tab would generate a predictable path in task space. Unfortunately, for a human the numbers that generate this motion can be difficult to visualize without the help of a computer. In addition, there was no way to predict what the final configuration of the robot would be.

Lack of feedback: In initial iterations of this GUI, there was no text box for feedback from the robot. This was later added for debugging purposes, and to provide some feedback for the current state of the motions. Unfortunately, the overall design of the GUI was not designed for feedback. In some cases, the settings displayed on the GUI were different from the settings the robot followed. During the Drill task at the DRC Trials, the setting for the "other hand state" was changed due to a misclick. This was not reflected on the GUI, and so was not corrected by the operator. The robot fell, and analysis after the event determined this setting was the cause. If the GUI had properly updated with the state of the robot, the fall would have been prevented.

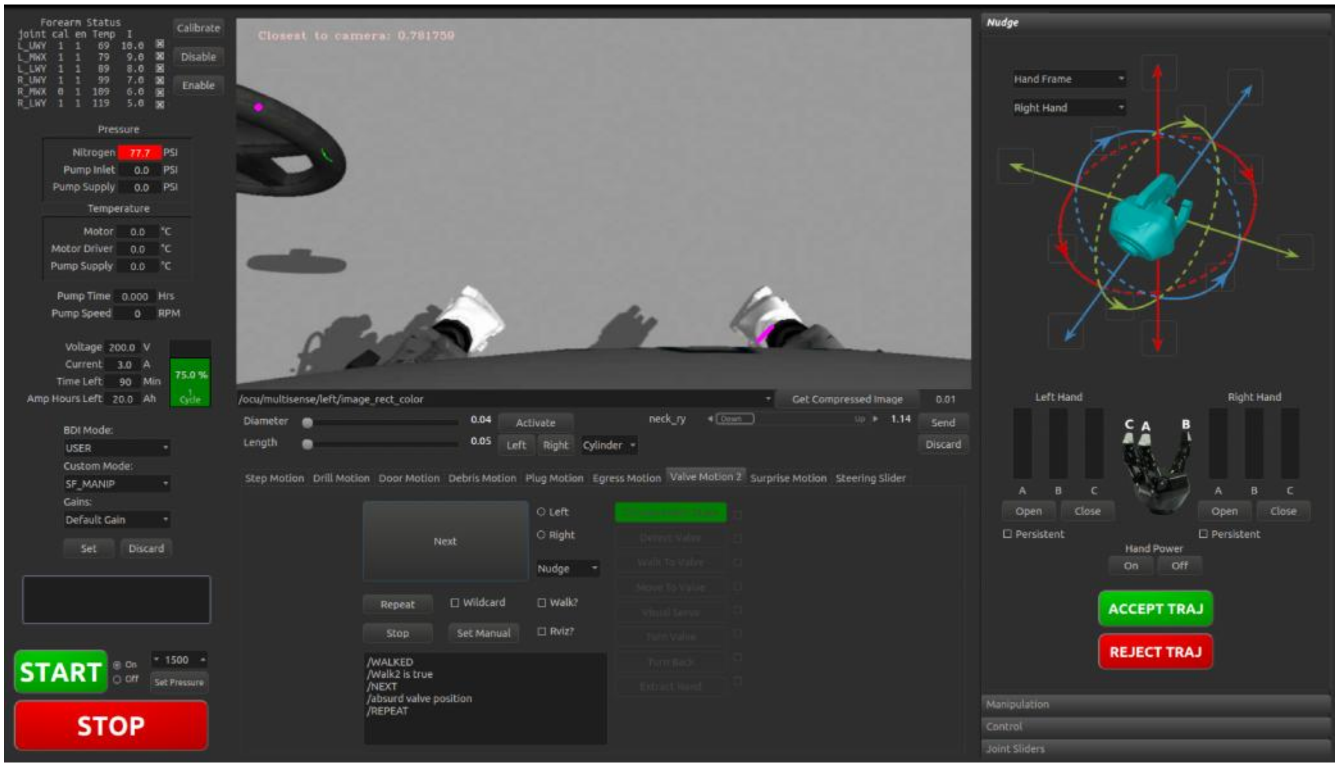
In addition to these drawbacks, it should be noted that this interface design was heavily dependent on the users. While the interface could calculate simple motions, the creation of more complex motions was only possible using the nudging feature, which was entirely controlled by the human operator.

Using these lessons learned, we developed a new interface for the DRC Finals

3.3 Final Design

The original GUIs were updated and replaced for the DRC Finals. Most of WGUI's features were kept for the final design, but were reorganized. In addition, WGUI featured tools for manipulation. Nudging was added as a permanent tool along with other hand controls. In addition, buttons were added to integrate the GUI with a motion planner.

The key addition to the GUI are the tabs at the bottom of the GUI. These are task specific, and are designed to manage state machines for each task. As mentioned earlier, for many tasks it was possible to preplan motions in advance rather than generating them on the fly. This takes it one step further and allows for sequences of actions to be planned in advance using a finite state machine. These tabs also solved the feedback issue – each panel updates the current state of state machine using feedback from the robot. As such, the settings and parameters of the robot's actions are known at all times, and are available to the operator.



```
  Forearm Status
joint cal en Temp  I
L_UWY  1  1   69  10.0
L_MWX  1  1   79   9.0
L_LWY  1  1   89   8.0
R_UWY  1  1   99   7.0
R_MWX  0  1  109   6.0
R_LWY  1  1  119   5.0
```

Closest to camera: 0.781750

Pressure
Nitrogen 77.7 PSI
Pump Inlet 0.0 PSI
Pump Supply 0.0 PSI

Temperature
Motor 0.0 °C
Motor Driver 0.0 °C
Pump Supply 0.0 °C

Pump Time 0.000 Hrs
Pump Speed 0 RPM

Voltage 200.0 V
Current 3.0 A
Time Left 90 Min
Amp Hours Left 20.0 Ah
75.0 % 1 Cycle

BDI Mode: USER
Custom Mode: SF_MANIP
Gains: Default Gain

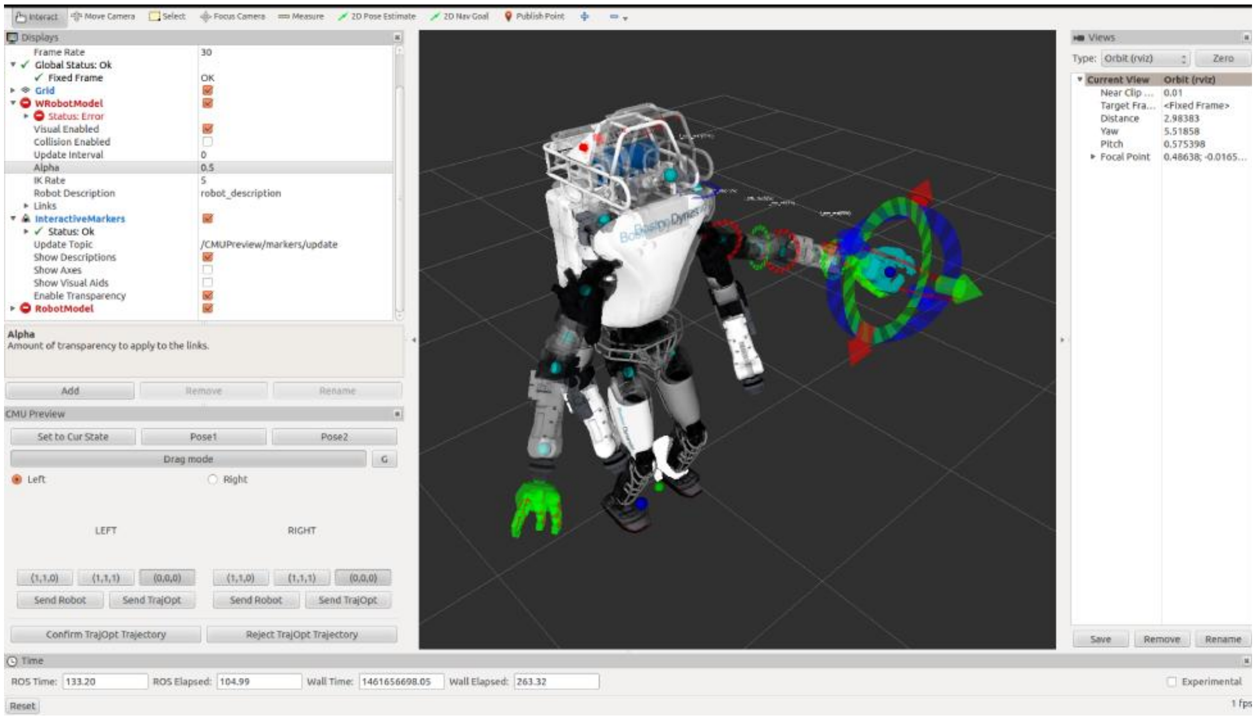
/ocu/multisense/left/image_rect_color
Diameter 0.04
Length 0.05
neck_ry 1.14
0.01
1500

Hand Frame
Right Hand

```
/WALKED
/Walk2 is true
/NEXT
/absurd valve position
/REPEAT
```

Figure 17: WGUI design for the DRC Finals. This is a modular GUI, allowing for new tools to be created as needed. On the left are various robot status interfaces. In the top center is the current camera image from the robot (currently displaying simulation data). On the right are tools for control of the hands, nudging, manipulation tasks, and joints. In the bottom center are tabs for each individual task and for walking commands.

However, WGUI does not provide for visualization. Instead, WGUI is run alongside RViz which contains an instance of CMU Preview, a visualization tool for our controller. CMU Preview runs an isolated version of our Inverse Kinematics solver, which allows operators to view how the robot will react to certain commands before they send them. It also makes it much clearer where the targets are.



```
Frame Rate              30
Global Status: Ok
  Fixed Frame           OK
Grid                    ✓
WRobotModel             ✓
  Status: Error
  Visual Enabled        ✓
  Collision Enabled     ☐
  Update Interval       0
  Alpha                 0.5
  IK Rate               5
  Robot Description     robot_description
  Links
InteractiveMarkers      ✓
  Status: Ok
  Update Topic          /CMUPreview/markers/update
  Show Descriptions     ✓
  Show Axes             ☐
  Show Visual Aids      ☐
  Enable Transparency   ✓
RobotModel              ✓

Alpha
Amount of transparency to apply to the links.
```

```
Type: Orbit (rviz)
Current View            Orbit (rviz)
  Near Clip ...         0.01
  Target Fra...         <Fixed Frame>
  Distance              2.98383
  Yaw                   5.51858
  Pitch                 0.575398
  Focal Point           0.48638; -0.0165...
```

```
ROS Time: 133.20   ROS Elapsed: 104.99   Wall Time: 1461656698.05   Wall Elapsed: 263.32
```

Figure 18: CMU Preview in Rviz. This tool allows visualization of manipulation commands. In this image, the "drag mode" has been enabled, which allows users to drag the robot target around in real time. Note that CMU Preview is contained within Rviz. Rviz is a free visualization tool that is provided with ROS. Currently also displayed in Rviz is the actual robot pose. The actual pose is solid, while the CMU Preview simulation is slightly transparent.

Shown in the image is a 3D draggable hand model (the teal hand with arrows and circles around it). This is only available if "drag mode" is enabled. By dragging this virtual hand around, operators can update the targets in real time and see how the robot will solve for those targets. Also note the circles around the arm. These allow users to individually control joints.

If drag mode is disabled, the robot will display simulations of commands sent to it. These commands are the same as those sent to the real robot, with the only difference being that the intended recipient of the command is CMU Preview and not the robot.

In both cases, once the user is satisfied with the motion, they can use the buttons on the left to send the command to the real robot.

In addition to CMU Preview, we use an additional visualization tool to view trajectories that are generated by Trajopt. This is provided with OpenRAVE (Diankov & Kuffner, 2008), a motion planning framework that is used by Trajopt. An example of the trajectory view in OpenRAVE is shown below.

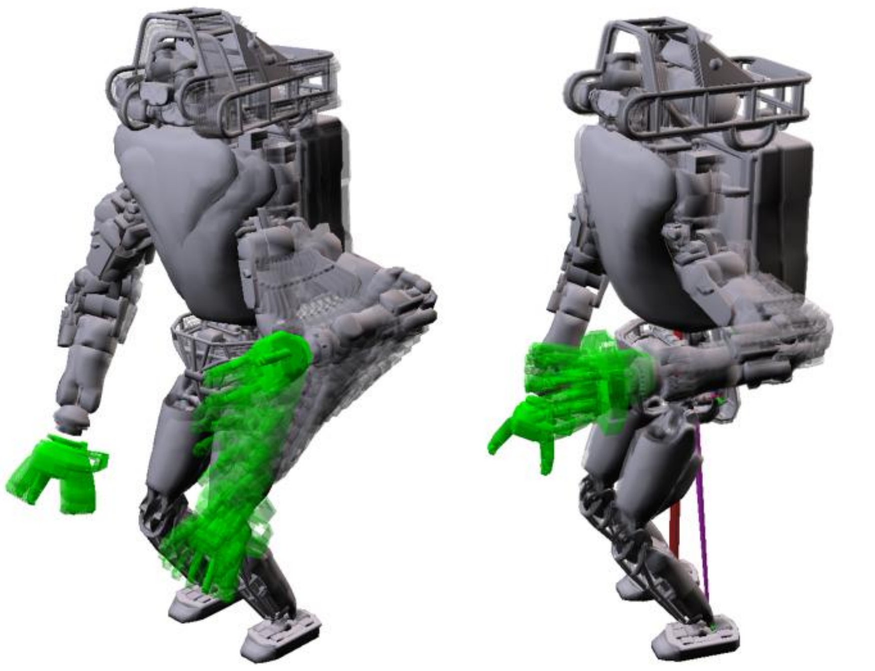


Figure 19: Trajopt generated trajectory. Left: a trajectory that raises the arm to a specified height and orienation (in this case for the valve task). Right: motion generated to insert the hand, rotate 360 degrees, and retact the hand. This was used for the valve task at the DRC Finals.

4 Motion Planning

In general, a robot cannot move freely in its environment, and is restricted to certain motions. This might be intrinsic to the robot (eg balance constraints, torque constraints, joint limits), or the restrictions could be due to the environment. In either case, it is often advantageous to plan a complete motion, rather than incrementally moving towards a target. This is the main difference between motion planning and controls.

Motion planning is concerned with generating trajectories that complete a goal while obeying any constraints. In the strictest sense, a trajectory is a continuous curve through configuration space that is parameterized by time. That is, for each point in time there is a corresponding configuration, resulting in a continuous motion as time increases. However, most trajectories are defined as sequences, rather than as a continuous function. For each waypoint, there is a timestamp that define when that configuration should be achieved. For times in between timestamps, the configuration is interpolated from the adjacent waypoints.

In many cases the goal is simply to achieve an endeffector position at the end of the motion. If there are no constraints on the robot's motion, this is simply an inverse kinematics problem. Usually, there are additional constraints. The most obvious is collision avoidance, which also happens to be one of the most difficult to solve. Other common constraints include: joint limits, torque limits, and path constraints. For Atlas (and humanoids in general), there is also a balance constraint, often formulated as a Zero Moment Point (ZMP) constraint.

4.1 Configuration Interpolation

The simplest "trajectory" is one that consists of two waypoints: the start configuration and an end configuration. The question then becomes how to interpolate between the two.

The easiest way to interpolate between points is a linear spline. This is written as below for motions from time *t*=0 to *t*=1:

$$q(t) = q_0(1-t) + q_1(t)$$

Where *q(t)* is the configuration *q* at time *t*, q_0 is the configuration at time *t*=0, and q_1 is the configuration at time *t*=1. This can be easily generalized to any starting and ending times.

One can also use more complex splines in configuration space, such as cubic splines or quantic splines. These splines allow the resulting function and some number of its derivatives to be differentiable, which translates to having smoother motions.

The main advantage of this interpolation in configuration space is its simplicity and numerical stability. Splines are governed by simple polynomial equations, and will always have solutions at every time *t*. However, there is no guarantee that for $0 < t < 1$ *q(t)* will not violate constraints, such as the ZMP constraint, even if q_0 and q_1 obey all constraints. Collision avoidance is also not guaranteed, and is typically worse as interpolation in configuration space can result in large sweeping motions.

Nonetheless, we still used interpolation in configuration space. Initially, we were only defining an initial configuration and ending configuration for a limited number of joints, typically a single arm at a time. We used Moveit to generate an IK solution for a particular pose *T*, then used that as our ending configuration. When executing the motion, only the arm would be interpolated. The other joints (legs, torso) were set by a whole body controller that ensured the ZMP constraint would not be violated, regardless of the arm's motion. In order to avoid collision, we would only use this technique in spaces that were free of obstacles.

Using configuration space interpolation like this does have a serious limitation, though. Because the interpolation only controls the arms, the torso and legs were free to move as long as the constraints were not violated. This meant that the kinematic chain from the feet to the end effector (which determine the end effector's position in the world) could change in undesirable ways. As such, the final end effector pose would often not match the initial pose *T* specified by the original problem. If the end effector pose needed to be accurate, one would have to control the arm after the motion is complete. We often used nudging to correct these errors.

This limits the usable cases for configuration interpolation. That is, we only used configuration interpolation when (a) there were no or very few obstacles that the robot could collide with and (b) when the final pose of the end effector could be inaccurate.

4.2 Task Space Interpolation

Once again, the simplest trajectory is one that is specified by a starting point and an ending point. However, unlike configuration space interpolation which operates in configuration space on configurations *q*, task space interpolation operates in task space on poses *x*. So for instance, a linear spline in task space would be:

$$x(t) = x_0(1-t) + x_1(t)$$

Where *x(t)* is the pose *x* at time *t*, x_0 is the pose at time *t*=0, and x_1 is the configuration at time *t*=1. For every time *t*, a corresponding configuration *q(t)* needs to be generated via IK.

Since a pose in task space only has 6 DoF, this give much more freedom to the system as humanoids have far more degrees of freedom (e.g. Atlas has 30 DoF assuming a fixed base). As such, there are many IK solutions that will satisfy *FK(q(t)) = x(t)*. This allows the IK to simultaneously solve for other constraints, which is exactly what we do. In this way, spline interpolation in task space can satisfy constraints for every time *t*. In addition, it is guaranteed to hit the task space target once the motion is complete, if the solution exists.

There are disadvantages to using task space interpolation. The most significant is the lack of solutions for every problem. Unlike configuration space interpolation, it is possible that for a given pose *x*, there is no solution to the IK problem. In fact, it is not guaranteed that between two feasible poses x_0 and x_1 that all poses *x(t)* have corresponding configurations. Take, for example, the situation of having one hand behind the back, and the target pose in front. Although it may be possible, it should be clear why this is a difficult task space trajectory to follow: the trajectory will need to pass through the torso.

Another facet of this issue is the lack of "continuity" in configuration space. Roughly speaking, for any given task space pose x_n achieved by q_n, there is no guarantee that a nearby point x_{n+1} has a nearby solution q_{n+1}. In essence, a small motion in task space may require a large motion in configuration space, e.g. motion in the null space to allow a continuous motion.

Collision avoidance is also not guaranteed for task space interpolation. However, unlike configuration space interpolation, task space interpolation will not generate sweeping motion, instead confining the end effector to a line. This may or may not help with collision avoidance, especially since only the end effector is confined to the line while the rest of the links are free to move around.

Nonetheless, the task space interpolation method was used extensively by our team. Since the path of the end effector could be accurately controlled, this was used for large motions where nudging was undesirable but the path was still important. For example, for the drill task, the straight line drilling motion was completed by linear task space interpolation.

4.3 TrajOpt

We decided to use TrajOpt for generating complex motion plans. TrajOpt is an optimization-based motion planner, and uses sequential convex decomposition to solve motion planning problems (Schulman, et al., 2013, June). It was originally developed by John Schulman at University of California, Berkeley.

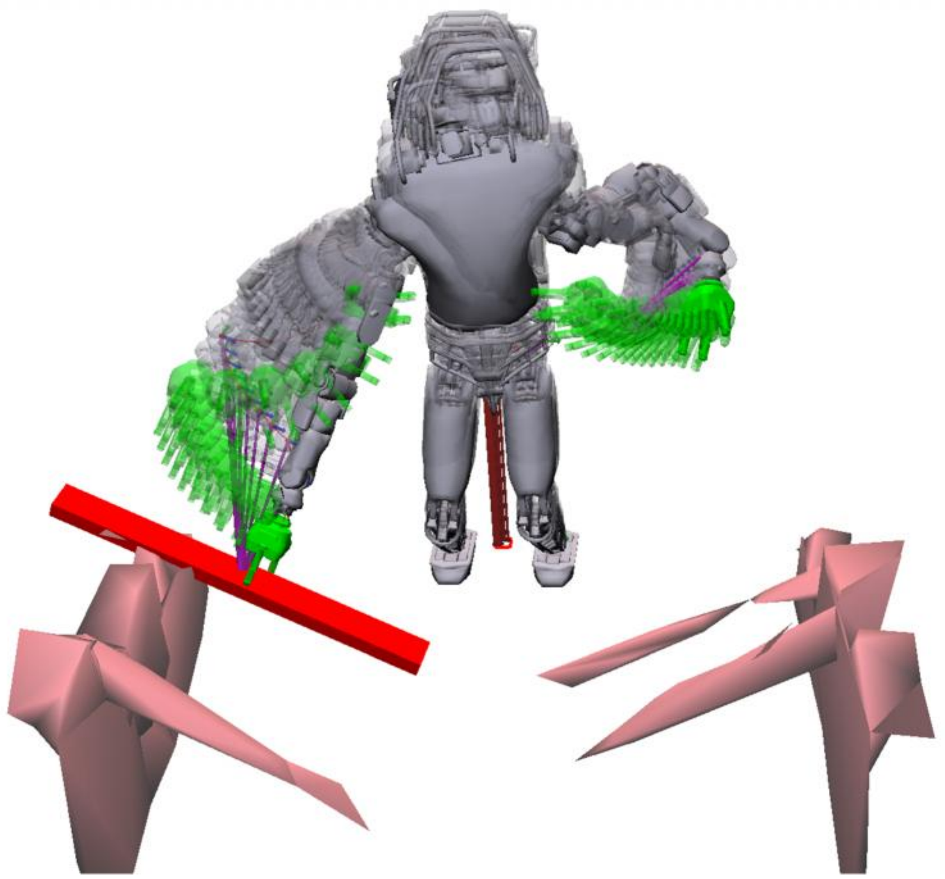


Figure 20: A motion plan generated by Trajopt. In this case, Atlas is reaching for a specified piece of wood. Note the constraints in the image: The last waypoint of the hand must reach the piece; the bar under the robot's pelvis indicates the ZMP constraint, and corresponds to the XY position of the CoM; the feet remain stationary during the action. Each duplicate atlas is a waypoint in the trajectory.

The optimization problem for trajectory generation is written as

$$\begin{aligned} &min\ f(x) \\ &s.t.\ g_i(x) \quad \leq 0\ for\ i = 1, 2, ..., n \\ &\quad\ h_i(x) \quad = 0\ for\ i = 1, 2, ..., m \end{aligned}$$

$$x = q_{1:T} = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_T \end{bmatrix}$$

That is, minimize the cost function *f(x)* subject to *n* inequality constraints $g_i(x)$ and *m* equality constraints $h_i(x)$. In this case, *x* is the entire trajectory written as a vector of *T* waypoints, with a full configuration q_i at each waypoint *i*. For Atlas, each configuration q_i has dimensionality 30 (we disregard the mobile base for manipulation planning), so for a trajectory with *T* waypoints, the full dimensionality of *x* is 30*T*.

Sequential convex decomposition works by solves optimization problems by formulating easily solvable convex problems at each step, progressing down the cost function similar to how gradient descent would. Constraints are handled by forming the Lagrangian seen below, treating the constraints as a cost, then solving the problem with progressively higher gains μ on the constraints.

$$min\ f(q) + \mu \sum_{i=1}^{n} |g_i(x)|^+ + \mu \sum_{i=1}^{m} |h_i(x)|$$

Note that $|a|^+$ is defined as *a* for *a>0*, and 0 for *a<0*. In other words, $|a|^+$ only has value when *a>0*.

Because of how Trajopt solves motion planning problems, it is neither complete nor globally optimal (although there is some guarantee of local optimality). In this case, optimal refers to the minima in the cost function used by Trajopt.

The actual structure of the problem varied between tasks, but most had the same general formulation. The problem can be separated into two different parts – the costs and the constraints. The cost function *f(x)* in the optimization problem summations of the various costs, while the constraints stay as separate functions. In addition, the costs can be generated with different weights, which make some costs more expensive that others.

The following are the functions we used.

4.3.1 Joint Displacement

For each timestep, the function is as follows:

$$f(q_{1:T}) = \sum_{i=1}^{T-1} ||q_i - q_{i-1}||^2$$

Where $q_{1:T}$ is the full trajectory, with q_i being the full configuration of the robot at waypoint *i*.

This is necessary for keeping the path as short as possible, as larger distances in configuration space are penalized. This is only used as a cost, as we want to minimize this distance but we have no limit on how long the path needs to be.

In some scenarios, it may be desirable to have the joint displacement cost only act on a subset of the waypoints. In this way, parts of the path will be more defined, as the waypoints are clustered closer together.

4.3.2 Collision Avoidance

Trajopt uses the Gilbert-Johnson-Keerthi (GJK) algorithm (Gilbert, Johnson, & Keerthi, 1988) to calculate distances between convex objects, and the Expanding Polytope Algorithm (EPA) (Bergen, 1999) to calculate penetration distance. Using these, the cost of collision can be changed into a distance based metric rather than a binary value (colliding or not colliding). This allows the gradient to be defined, and for minimum distance constraints to be set. For a more detailed explanation, see (Schulman, et al., 2013, June).

Given a point cloud generated by the Lidar, we run Hierarchical Approximate Convex Decomposition (HACD) to generate a collection of convex shapes (Mamou & Ghorbel, 2009). All of the convex hulls are used for collision checking.

Trajopt also approximates the swept volume between timesteps by using the convex hulls of body links. This is the correct swept volume for links that only translate, but in almost every situation a link undergoes translation and rotation. While there is a way to find the maximum distance of the error between the convex hull and the swept volume, we ignore this error as it is normally small.

This is normally used as a cost. When used as a constraint, it can restrict the problem too much. This is especially true for grabbing objects, where often contact or near contact with obstacles is desired.

4.3.3 Joint Position

This is formulated as

$$
\begin{aligned}
f(q) &= ||q - q_{target}||^2 \\
g(q) &\leq a(q - q_{limit}),\ \ a \in \{-1, 1\} \\
h(q) &= (q - q_{target})
\end{aligned}
$$

f(q) is the cost function, *g(q)* is the inequality constraint function, and *h(q)* is the equality constraint function. In *g(q)*, *a* is used to specify whether the value q_{limit} is an upper or lower limit. Note that for these functions, *q* is only a single configuration. When used for computing a full trajectory, the specific waypoints q_i are used.

In some cases, we needed the robot to have a particular joint in a desired configuration. For example, for the valve task we needed the wrist to have maximum range of motion, in order to turn the valve as much as possible. Therefore we added a constraint to push the wrist to its joint limits.

This is also used for enforcing joint limits when used as an inequality constraint.

This is used as both a constraint and a cost, depending on the situation.

4.3.4 Task-space Pose

This calculates the distance a link is from a desired pose for a given configuration;

$$f(q) = ||FK(q) - x_{target}||^2$$
$$h(q) = (FK(q) - x_{target})$$

f(q) is the cost function, *h(q)* is the equality constraint function, and *FK(q)* is the forward kinematics function that gives a pose in task space from a configuration *q*. Note that for these functions, *q* is only a single configuration. When used for computing a full trajectory, the specific waypoints q_i are used. Additionally, it is possible to formulate inequality constraint functions for task-space poses. When used as inequality constraints, the allowed space resembles that of a half-plane.

This is the canonical motion planning goal, that is, a problem with a target in task space. We use these for all tasks. In some cases, the target is in relation to a manipulation item or task. For all cases, we also use the pose constraint to enforce foot location, ie to avoid forcing the feet to slip.

For "goals", where the pose target is a desired position during a manipulation task, the function is used as a constraint; the goal must be met. The feet location is also set as a constraint, since if it fails we will face unknown friction forces and possibly move the base of the robot in an random direction, not to mention falling. We also use this to normalize the torso and pelvis, mainly in orientation (ie keep the torso upright), but as a cost instead of a constraint.

4.3.5 Zero Moment Point

This calculates a cost for Zero Moment Point. It consists of a series of half-planes in task space which confine the center of mass position. The half-planes' intersection with each other and the ground plane form the support polygon, or the convex hull of the foot contacts.

This is necessary to avoid falling, assuming a "static" motion where accelerations and velocities do not play a large part. In simpler terms, this keeps the center of mass above a support polygon, thus preventing falling. Partly for safety and partly for known but unfixable errors in the robot model, we restrict the support polygon to a very small box between the feet.

This is always used as a constraint during manipulation tasks.

5 Integration

While manipulation was an integral part of the entire system, it was by no means the sole factor in determining our success. Locomotion/walking and perception played important roles during the competition, and would be vital for any activity in an unstructured environment. This section will describe the integration between these different topics in humanoid research.

5.1 Perception

Perception is the robot's ability to detect and interpret its surrounding. Common examples of research that falls under perception are SLAM and object recognition. Manipulation and perception are obviously tied together, as perception can find objects of interest in the environment, which become the targets of manipulation techniques. Another tie between the two that is less apparent is the ability of manipulation to enhance perception of the environment.

We used object recognition to identify entities in the environment we cared about. For instance, we used a point cloud matching algorithm to identify the valve during the valve task. Once the valve was identified, its location was used to generate a motion in Trajopt that would turn the valve.

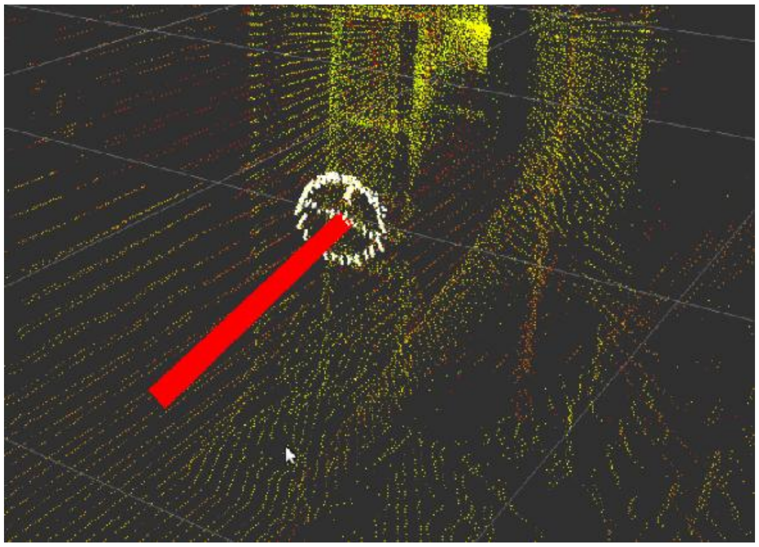


Figure 21: Point cloud with detected valve. The valve is marked by the red bar, which gives its axis of rotation and center location.

Similar methods were used for opening the door. The door handle location and orientation was identified, which were then sent to Trajopt to generate motion to open the door. This approach is detailed in (Banerjee, 2015).

There is also a possibility for manipulation to aid perception. This is demonstrated in one of the methods we devised for picking up debris, when we mounted a depth camera to the hand. An initial location for debris pieces was generated by a point cloud matching algorithm from the Multisense head. Using this initial pose, the arm was positioned over the debris to give the depth camera an unobstructed, close view of the debris. Thus, the localization for the hand in relation to the debris was significantly improved by relocating the hand with the depth camera. This in turn was used to improve manipulation via visual servoing.

Perception for the DRC was also supplemented by human interaction. For instance, the valve detection was aided by a human, who would provide an initial guess for the perception algorithm to search. This significantly reduced the difficulty in identifying objects.

In a similar vein, teleoperation can be seen as a combination of perception and manipulation. In this case, the perception is performed entirely by the human operator. In the case of nudging, the human also performed much of the manipulation task. This can be seen in the surprise task, where the human operator used the camera image and point cloud data to guide the hand.

5.2 Locomotion

Locomotion and manipulation are closed tied together when a robot operates in an unstructured environment. A key component of the manipulation problem is the location of the robot. If the robot is not properly positioned, manipulation becomes difficult or impossible. As an example, if the robot is too far away from its target, it will not be able to reach its target. Conversely, if the robot is too close to the location of the the target, the robot may not be able to effectively work. This positioning is based on the workspace of the robot, which is a function of how effectively the arm can move in space. A heat map of the Atlas v3's workspace can be found in (Fallon, et al., 2015).

In our implementation, the perception systems would identify an object of interest and generate a final foot location for the robot to move to. The walking controller would then move the robot to that location, where manipulation could be performed. The appropriate locations for the footsteps were determined empirically through trial and error.

6 Results

This section will go over the results from the DRC that pertain to the manipulation tasks.

6.1 DARPA Robotics Challenge, Trials

At the DRC Trials, we placed 7th out of 16 teams, scoring 11 points out of 24. The following are brief statements on the results for each task. A more complete description of the results can be found in (DeDonato, et al., 2015).

6.1.1 Debris

The debris task was unsuccessful. We were very close to picking up a 5th piece of debris, which would have awarded the 1st point. Unfortunately, our robot contacted the surrounding wall and fell before the point was scored. There was not enough remaining time to continue.

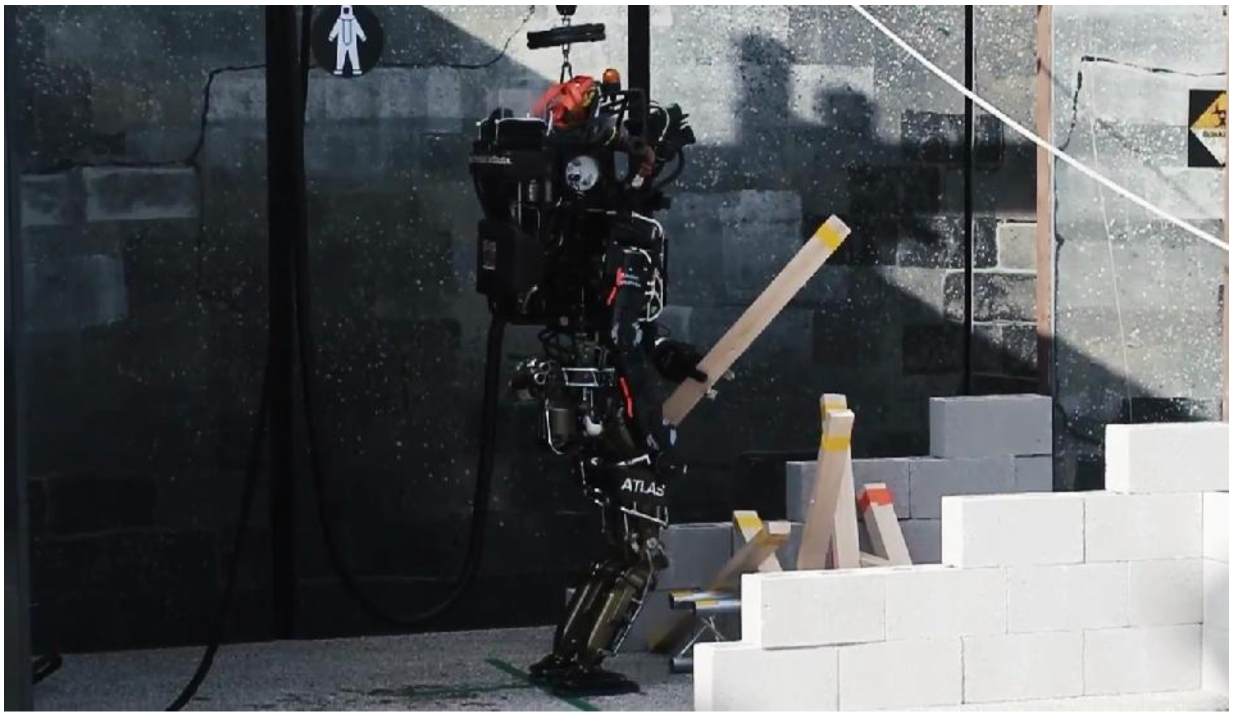


Figure 22: The DRC Trials Debris Task. For every 5 pieces of debris retrieved, 1 point is awarded.

6.1.2 Door

The door task was unsuccessful, albeit do to unforeseen circumstances. The Trials took place outdoors, and as such was exposed to the wind. After opening the first door, we turned in order to pass through. Unfortunately, during the maneuver the door was closed by the wind. We did not anticipate this, and had no working strategy to either keep the door open during motion or the reopen the door after it closed.

6.1.3 Wall

The wall task was unsuccessful, despite being one of our most reliable tasks during testing. The main cause for the failure was human operator error and poor GUI design. During the task, the operator accidentally put the empty hand (left hand) into hold position mode, severely limiting the workspace of the right arm. Eventually, the robot was pushed into a strange configuration that ended in a fall.

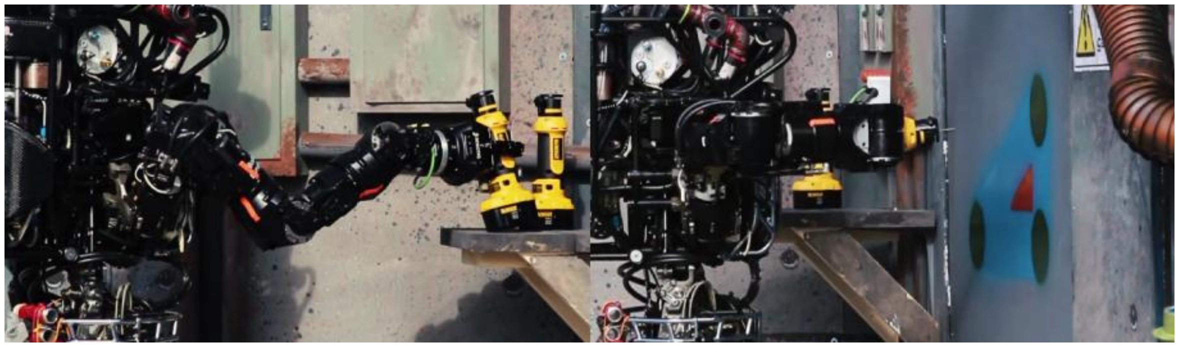


Figure 23: DRC Trials Wall task. Left: Atlas retreiving a drill. Right: Atlas preparing to drill the wall, shortly before falling.

6.1.4 Valve

The valve task was performed as expected. We successfully turned all three valves without falling, although we were a tad closer to falling that we would have liked. This is primarily due to having inconsistent positioning of the robot prior to manipulating the valves; the location of the robot was chosen by the human operators for each run, leading to inconsistencies.

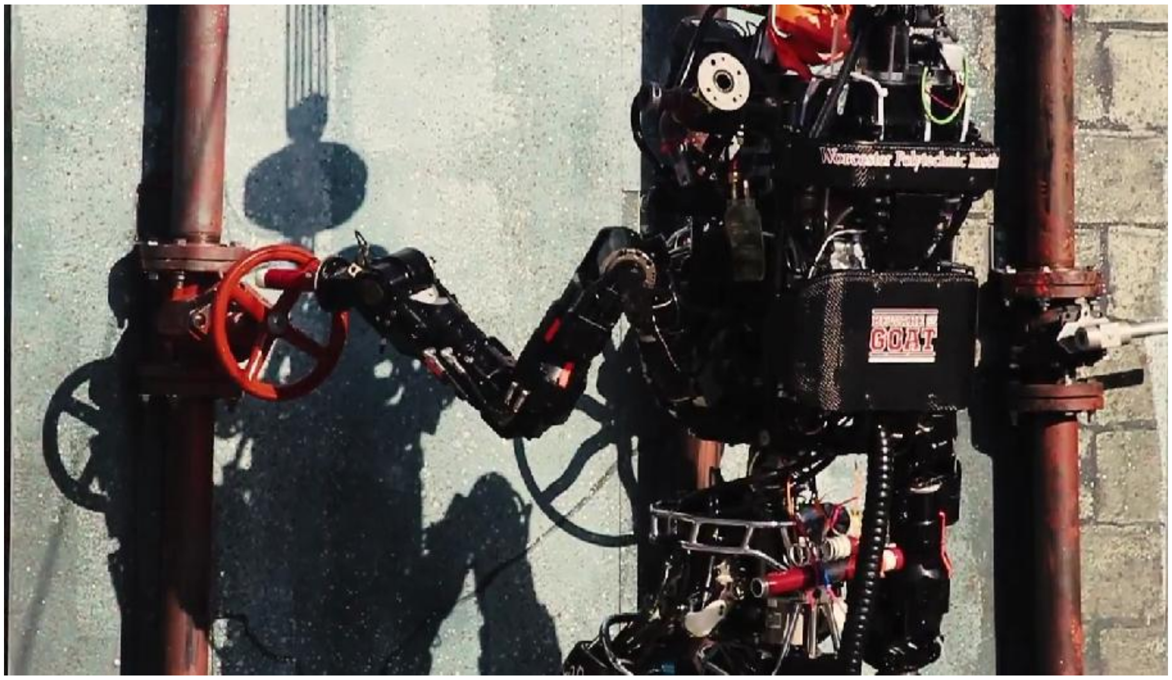


Figure 24: DRC Trials Valve task. Atlas is manipulating one of three valves.

6.1.5 Hose

The hose task was performed as expected. We were able to grab the hose and move it to the y-pipe, but were unable to attach the hose; we were awarded 2 out of 3 points. We had never attached the hose to the pipe in testing.

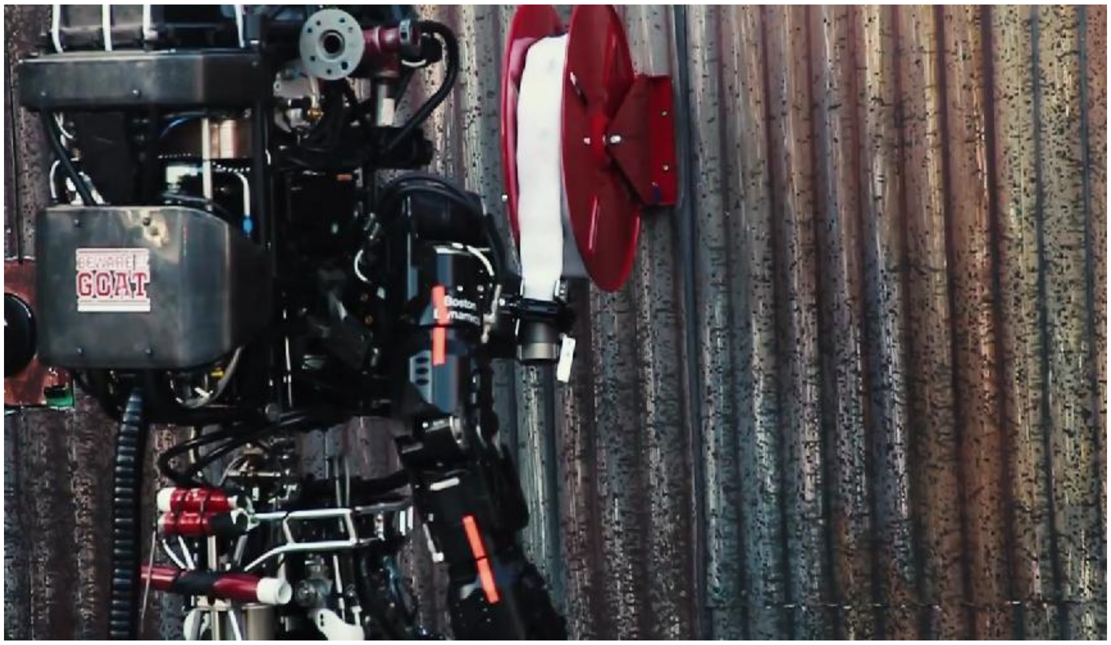


Figure 25: DRC Trials Hose task. Atlas retrieving the hose.

6.2 DARPA Robotics Challenge, Finals

The DRC Finals took place on June 5 and 6, 2016. We ran the robot through the course twice, once on each day. In addition, on June 4 we ran a rehearsal run.

At the DRC Finals, we placed 7th out of 23 teams, and scored 7 points out of 8 in 56 minutes. In addition, we were the only team that did not fall and/or need to reset (NREC's CHIMP fell over, but did not need to be reset). Overall, the robot performed very well, and our algorithms also worked as desired. The only task we did not complete (in either day) was the drill task. The following are brief statements on the results for each manipulation task. A more complete description of the results can be found in (Atkeson et al, 2015).

6.2.1 Door

The door task was performed as expected. On both days, the robot successfully passed through the door. However, during one of the rehearsal runs, the robot was caught in the doorway and needed to be manually extracted.

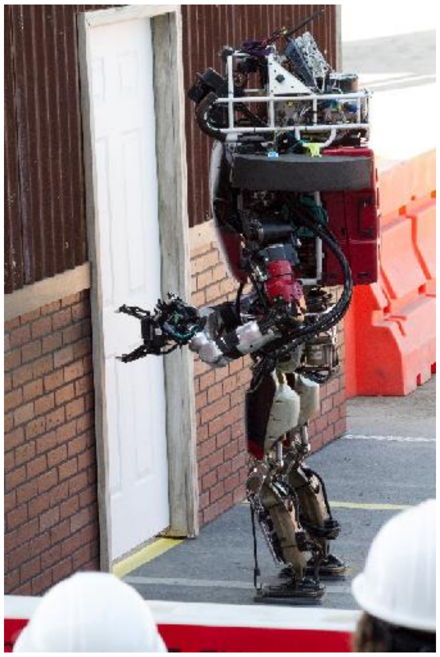


Figure 26: DRC Finals Door Task. Atlas reaching for the door.

6.2.2 Valve

The valve task was performed as expected. Both days, the robot successfully turned the valve for full points. The valve was detected as expected, and afterwards the robot executed all of the steps for the valve task without human intervention. That is, it walked to the valve, planned and executed a motion to turn the valve, and stepped back.

6.2.3 Drill

The drill task was failed on both days, despite being very reliable during testing. On the first day, a bug in our software caused the robot to attempt to reposition itself after grabbing the drill, which included a command to open the hand, thus dropping the drill. The bug was identified and fixed by the second day. On the second day, the right forearm failed during the task, going limp. Although this may have caused a fall or other fatal event, our operators were able to detect the problem and stop the robot. The drill was dropped, and we moved onto the next task.

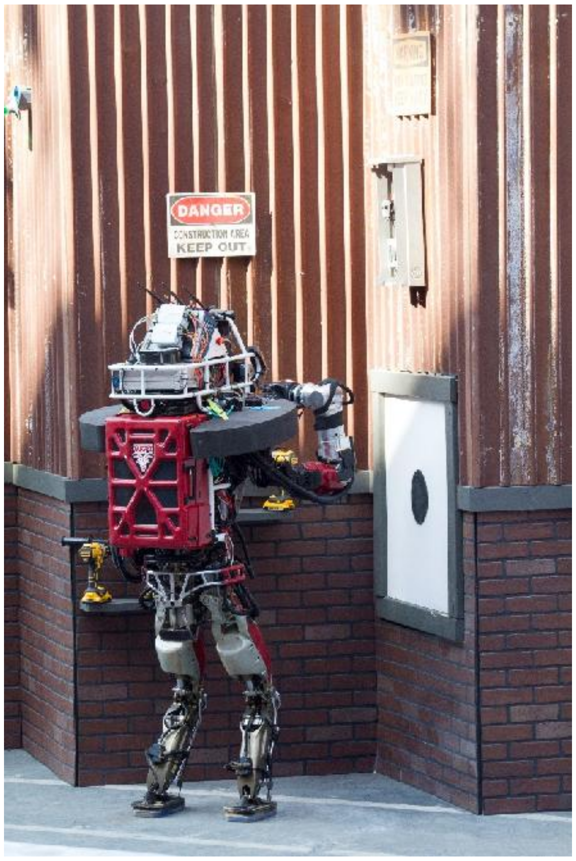


Figure 27: DRC Finals Drill task. Atlas preparing to pick up the drill. Note the sheetrock wall on the right.

6.2.4 Surprise

The surprise task was completed both days. This was also one of the fastest tasks. The task actually changed between the two days: on the first day, we were tasked with operating a lever, while on the second day we needed to move a plug from one socket to the other. On both days, we used teleoperation to complete the task.

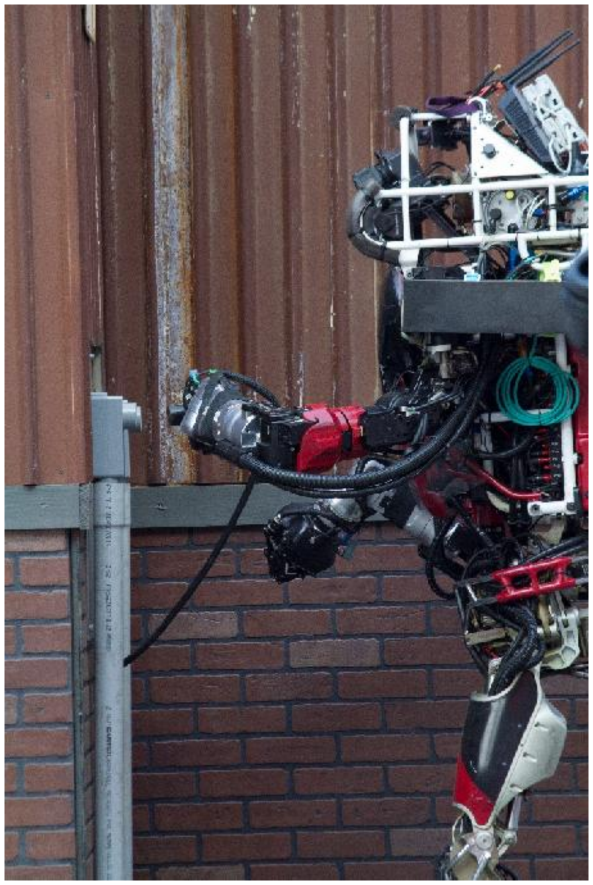


Figure 28: DRC Finals surprise task. Atlas is moving the plug from one socket to the other.

7 Future work

Despite these advances, there is still much work to be done. Below are a few topics we could have improved on.

Dynamic motions: Most of the motions we took were quasi-static, or slow enough that velocities and accelerations did not play a large part. While this worked for the DRC, it resulted in very slow motions. Dynamic motions, on the other hand, would be much faster. However, at the speeds where the motion could be considered “dynamic”, momentum and forces play a large enough part that they can cause falls. Nonetheless, the additional speed would allow for reactive movements: that is, rapid movements that can avoid perturbations in the environment, such as moving obstacles.

Increased force: along the same vein as the above, we were not able to apply large forces to the environment. Because the constraints for balance were largely dependent on foot pressures and the ZMP CoM constraint, additional forces in the hands were not accounted for. As such, when the hand encountered large forces, such as pushing against a wall, the dynamic model did not account for this and so would not prevent a fall. Future work should include the addition forces in the dynamic model and account for them when balancing.

Virtual reality: Although our interface performed very well at the DRC Finals, there can still be improvements made to it. Virtual reality provides for a level of immersion not possible for our system. This can allow the operators to more effective interpret the environment, and therefore enable them to better interact with the environment.

8 Conclusion

We have developed effective techniques for manipulation on humanoid robots. All of our algorithms and methods were applied on the Atlas robot during the DARPA Robotics Challenge. The techinques we created were designed with human-supervision in mind, which allowed for complex tasks such as perception and high-level goal planning to be offloaded to the human operator.

In particular, we developed an interface for manipulation that was used by our operators. This interface was developed in two parts, first for the DRC Trials and then the DRC Finals. Our initial GUI placed too much responsibility in the hands of the operator, with no options for automatically generating complex motions. Following lessons learned from the DRC Trials, we redesigned the interface to for the DRC Finals.

The final design incorporated far more autonomous and automatic planning. Manipulation in the DRC finals typically followed a pre-planned set of actions, greatly reducing the cognitive load on the operators. This meant the operators could spend more time verifying the safety of the robot actions and watching for disturbances or dangerous edge cases. However, the interface still allowed for enough direct operator control that if necessary, a human could teleoperate the robot. This combination led to significantly better performance during the DRC Finals compared to the DRC Trials.

In addition, we implemented Trajopt in our system, which allowed for the automatic generation of complex motions. This allowed our robot to act more autonomously, although we still relied on human operators to accept or reject trajectories generated by Trajopt as a failsafe against bad robot motions. Trajopt became an integral part of the manipulation tool set, and was used in almost every task.

When these manipulation tools were integrated with the rest of the robot system, our Atlas was given the ability to effectively operate in the outdoors environment of the DRC. While many of the actions were specific to tasks in the DRC, the methods used to complete the tasks are largely independent of the DRC (for instance, the tools for turning a valve could be applied to any valve). In this way, we have created methods and techniques for manipulation in unstructured environments, albeit with high-level known goals.

9 References

Astrom, K. J., & Hagglund, T. (1995). *PID controllers: Theory, design and tuning.* Research Triangle Park, NC: ISA: The Instumentation, Systems, and Automation Society.

Atkeson, C. G., Babu, B. P., Banerjee, N., Berenson, D., Bove, C. P., Cui, X., . . . Xinjilefu, X. (2015). No falls, no resets: Reliable humanoid behavior in the DARPA robotics challenge. *Humanoids.*

Banerjee, N. (2015). Human-Supervised Semi-Autonomous Approach for the DARPA Robotics Challenge Door Task. *Worcester Polytechnic Institute Dissertation*.

Bergen, G. v. (1999). A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools, 4*(2), 7-25.

DeDonato, M., Dimitrov, V., Du, R., Giovacchini, R., Knoedler, K., Long, X., . . . Atkeson, C. G. (2015). Human-in-the-loop Control of a Humanoid Robot for Disaster Response: A Report from the DARPA Robotics Challenge Trials. *Journal of Field Robotics, 32*(2), 275-292. doi:10.1002/rob.21567

Diankov, R., & Kuffner, J. (2008). Openrave: A planning architecture for autonomous robotics. *Roobtics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34, 79.*

Einicke, G. A. (Ed.). (2012). *Smoothing, Filtering and Prediction - Estimating the Past, Present and Future.* InTeO.

Fallon, M., Kuindersma, S., Karumanchi, S., Antone, M., Schneider, T., Dai, H., . . . Koolen, T. (2015). An Architecture for Online Affordance-based Perception and Whole-body Planning. *Journal of Field Robotics, 32*(2), 229-254.

Feng, S., Whitman, E., Xinjilefu, X., & Atkeson, C. G. (2015). Optimization-based Full Body Control for the DARPA Robotis Challenge. *Journal of Field Robotics, 32*(2), 293-312.

Geraerts, R., & Overmars, M. H. (2004). A comparative study of probabilitic roadmap planners. *Algorithmic Foundations of Robotics V*.

Gilbert, E. G., Johnson, D. W., & Keerthi, S. S. (1988). A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation, 4*(2), 193-203.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of basic Engineering, 82*(1), 35-45.

LaValle, S. M., & Kuffner, J. J. (2000). Rapidly-exploring random trees: Progress and prospects.

Mamou, K., & Ghorbel, F. (2009). A simple and efficient approach for 3d mesh approximate convex decomposition. *16th IEEE International Conference on Image Processing (ICIP V9)*, (pp. 3501-3504).

Pratt, G., & Manzo, J. (2013). The darpa robotics challenge [competitions]. *IEEE Robotics Automation Magazine, 20*(2), 10-12.

Pratt, J., Carff, J., Drakunov, S., & Goswami, A. (2006, December). Caputre point: A step toward humanoid push recovery. *Humanoid Robots, 2006 6th IEE-RAS International Conference on* (pp. 200-207). IEEE.

Ratliff, N., Zucker, M., Bagnell, J. A., & Srinivasa, S. (2009, May). CHOMP: Gradient optimization techniques for efficient motion planning. *Robotics and Automation, 2009. ICRA'09.* IEEE.

Ribeiro, M. I. (2004). Kalman and extended kalman filters: Concept, derivation and properties. *Institute for Systems and Robotics, 43*.

Schulman, J., Ho, J., Lee, A., Awwal, I., Bradlow, H., & Abbeel, P. (2013, June). Finding Locally Optimal, Collision-Free Trajectories with Sequential Convex Optimization. *Robotics:science and systems, 9*(1), 1-10.

Wan, E. A., & Van Der Merwe, R. (2000). The unscented Kalman filter for nonlinear estimation. *Adaptive Systems for Signal Procesing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000*, 153-158.

Xinjilefu, X. (2015). State Estimation for Humanoid Robots. *(No. CMU-RI-TR-15-20).* CARNEGIE MELLON UNIV PITTSURGH PA ROBOTICS INST.

10 Appendix A

EW_MANIP API:

Communicate by sending atlas_ros_msgs::field_param

command line:

```
rostopic pub /atlas/field_params_cmu_walk atlas_ros_msgs/field_param “{cmd : [1], param : [0, 0, 0]}” -1
```

Keyboard Nudge Interface:

While standing still and not executing another command, it will respond to a keyboard interface. Each key tap will nudge the hand one centimeter.

Up Arrow: Z+=0.01

Down Arrow: Z-=0.01

Left Arrow: Y+=0.01

Right Arrow: Y-=0.01

i “in”: X-=0.01

o “out”: X+=0.01

e: pitch up

d: pitch down

s: yaw left

f: yaw right

w: roll left

r: roll right

q: neck up

a: neck down

It will move left, right, or both hands according to the last motion command set. If you wish to change the active the active hand, send a mode 0 command with 0 motion. Neither hand will be active at controller startup.

For 1 hand motion, rotations are in hand coordinates. Sending capital letters puts them in world coordinates. For 1 hand motion, rotations are always in world coordinates, and they rotate the relative hand position.

To start our keyboard interface run:

```
rosrun send_field_param eric_teleop
```
and type with that terminal active.

To send your own nudge commands, the format is:

cmd[0]: 17

cmd[1]: which hand

- default: same as previous command
- 10 - left
- 11 - right
- 12 - both (unimplemented; tell me if you want it)

cmd[2]: direction

- 105: in
- 111: out
- 65: up
- 66: down
- 68: left
- 67: right

cmd

cmd[0]: Mode

- 0/default - Relative Linear Motion
- 1 - circle
- 2 - freeze

3 - plun

4 - load parameters

5 - direct joint control

6 - body motion

7 - neck command

8 - put MBY joint against hard stop

9 - return MBY joint to normal configuration

10 - set hand location

11 - pre-defined arm pose

12 - Set Free Rotation Axis

13 - set translational nudge increment

14 - set rotational nudge increment

17 - keyboard commands

param (mode determined by cmd[0])

Relative Linear Motion Mode (default/cmd[0] = 0):

Command a relative motion in world coordinates. (World coordinates are x-forward, y-left, z-up defined based on the direction the robot is facing when my controller is started.)

cmd[1]: which hand is being commanded

0 - LEFT

1 - RIGHT

default - BOTH

cmd[2]: allow axial rotation on

0 - LEFT

1 - RIGHT

2 - BOTH

default - NEITHER

cmd[3]: what do do with the other hand (only active if cmd[1] is 0 or 1)

0 - maintain current position (default)

1 - lock joints

param[0]: dX

param[1]: dY

param[2]: dZ

param[3]: T

param[4]: dRoll (default: 0)

param[5]: dPitch (default: 0)

param[6]: dYaw (default: 0)

Circle Mode (cmd[0] = 1):

Move hand in a circle based on a center point {cX, cY, cZ} and the IK hand axis

cmd[1]: which hand is being commanded

0 - LEFT

1 - RIGHT

default - BOTH

cmd[2]: allow axial rotation

0 - Yes

1 - No (default)

cmd[3]: what do do with the other hand

0 - maintain current position (default)

1 - lock joints

cmd[4]: angle mode

0/default - whichever way the hand is pointing (yaw only)

1 - user specified (param[5]

param[0]: cX

param[1]: cY

param[2]: cZ

param[3]: T

param[4]: Total rotation angle in radians (positive is clockwise)

param[5]: Valve angle; 0: in the Y-Z plane; positive rotates clockwise; ignored if cmd[4]!=1

Freeze Mode (cmd[0] = 2):

Stop whatever you are doing, fix all arm and spine joints, and simply balance.

Plunge/Retract Mode (cmd[0] = 3):

Move the hand along the hand axis. This is intended for removing the hand smoothly from the valve.

cmd[1]: which hand is being commanded

0 - LEFT

1 - RIGHT

cmd[2]: allow axial rotation

0 - Yes

1 - No (default)

cmd[3]: what do do with the other hand

0 - maintain current position (default)

1 - lock joints

param[0]: motion in the hand axis direction (positive is plunge; negative is retract)

param[1]: T

Load Parameters Mode (cmd[0] = 4):

Load constants from configuration file.

Joint Control Mode (cmd[0] = 5):

Directly command joint angles in one of the arms.

cmd[1]: which hand is being commanded

0 - LEFT

1 - RIGHT

cmd[2]: what do do with the other hand (only active if cmd[1] is 0 or 1)

0 - maintain current position (default)

1 - lock joints

param[0]-param[5]: the six arm joints (starting at the shoulder)

param[6]: duration of move (optional: default = 2.0)

Body Motion Mode (cmd[0] = 6):

Adjust the desired body position and orientation. Both position and orientation commands are relative. Orientation is for both the pelvis and torso.

ADJUST THE COM X AT YOUR OWN RISK

cmd[1]: Whether hands move with you

0 - Hands remain stationary (default)

1 - Hands translate/rotate with body (constant arm angles)

param[0]: motion of the CoM in the X direction (positive is forward) param[1]: motion of the CoM in the Y direction (positive is left)

param[2]: motion of the pelvis in the Z direction (positive is up)

param[3]: T

param[4]: dRoll (default: 0)

param[5]: dPitch (default: 0)

param[6]: dYaw (default: 0)

Neck Command (cmd[0] = 7):

Command the neck angle.

param[0]: neck angle

Put MBY Joint Against Hard Stop (cmd[0] = 8):

Return MBY Joint to Normal Configuration (cmd[0] = 9):

Set Hand Rotation (cmd[0] = 10):

Specify the distance between the last wrist joint and the hand point. Rotation of the hand will occur about this point.

param[0]: distance between “hand” and the final wrist joint

Pre-Defined Arm pose (cmd[0] = 11):

Move both arms to a pre-defined hard-coded pose.

cmd[1]: which pose

Set Free Rotation Axis (cmd[0] = 12):

Set which axis the hand is allowed to freely rotate about (supposing axial rotation is on).

cmd[1]: which axis

0 - normal hand axis (starting case)

1 - drill axis

Set Translational Nudge Increment (cmd[0] = 13):

Set the increment for nudging (in meters) translationally. (starts at 0.01 m)

param[0]: increment

Set Rotational Nudge Increment (cmd[0] = 14):

Set the increment for nudging (in radians) rotationally. (starts at 0.02 rad)

param[0]: increment