AFRL-RI-RS-TR-2015-264

TRUSTED REMOTE OPERATION OF PROXIMATE EMERGENCY ROBOTS (TROOPER): DARPA ROBOTICS CHALLENGE

LOCKHEED MARTIN ADVANCED TECHNOLOGY LABORATORIES

DECEMBER 2015

FINAL TECHNICAL REPORT

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2015-264 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /	**/ S /**
ROGER J. DZIEGIEL, JR	MICHAEL J. WESSING
Work Unit Manager	Deputy Chief, Information Intelligence Systems and Analysis Division
	Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* DEC 2015	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* FEB 2012 – NOV 2015

4. TITLE AND SUBTITLE TRUSTED REMOTE OPERATION OF PROXIMATE EMERGENCY ROBOTS (TROOPER): DARPA Robotics Challenge	**5a. CONTRACT NUMBER** FA8750-12-2-0311
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62702E
6. AUTHOR(S) Steven Gray, Robert Chevalier, Nicholas DiLeo, Aron Rubin, Benjamin Caimano, Kenneth Chaney II, Todd Danko, Michael Hannan, David Kotfis, Daniel Donavanik, Alex Zhu	**5d. PROJECT NUMBER** ROBO
	5e. TASK NUMBER PR
	5f. WORK UNIT NUMBER 02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lockheed Martin Advanced Technology Laboratories, 3 Executive Campus, Suite 600, Cherry Hill, NJ 08002; University of Pennsylvania, 220 S. 33rd Street, Philadelphia, PA 19104; Rennselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIED, 525 Brooks Road, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2015-264

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Recent robotics efforts have led to automating simple, repetitive manipulation tasks to speed up execution and lessen an operator's cognitive load, allowing them to focus on higher level objectives. However, the robot will eventually encounter something unexpected, and if this exceeds the tolerance of automated solutions there must be a way to fall back gracefully to teleoperation. To address this challenge, we present our human-guided autonomy solution in the context of the DARPA Robotics Challenge (DRC) Finals. We describe the software architecture that Team TROOPER developed and used on an Atlas humanoid robot. Our design emphasizes human-on-the-loop control where an operator simply expresses a desired high level goal for which the reasoning component assembles an appropriate chain of subtasks. We employ perception, planning, and control automation for execution of subtasks. If subtasks fail, or if changing environmental conditions invalidate the planned subtasks, the system automatically generates a new chain. The operator is also able to intervene at any stage of execution, enabling operator involvement to increase as confidence in automation decreases. We present our performance at the DRC Finals as well as lessons learned.

15. SUBJECT TERMS
collaborative autonomy, human-guided autonomy, emergency response, legged robots, mobile manipulation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** ROGER J. DZIEGIEL, JR
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	SAR	99	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 SUMMARY

The motivation for the DARPA Robotics Challenge has been to develop robots and software capable of competent, semi-autonomous performance of tasks in a disaster scenario. Our experience on the DARPA Robotics Challenge began with the Virtual Robotics Challenge, where we wrote the software to allow a simulated Boston Dynamics Atlas robot to navigate rough terrain, drive a vehicle, and manipulate both a hose and valve. We placed 8th in the competition held in June 2013, qualifying us to receive a physical Atlas robot. Four months after receiving the Atlas robot, we competed in the DRC Trials in December 2013. Here, the robot had eight tasks: turning valves, opening doors, cutting drywall, driving a vehicle, climbing a ladder, manipulating a hose, handling debris, and traversing rough terrain. Again, we placed 8th and secured a funded spot for the 2015 DRC Finals. For the Finals, the robot had to complete simplified versions of the previous tasks in one continuous, hour-long run. We designed and implemented a system that is autonomous, though with the ability to ask a human operator for guidance when necessary. Lack of robustness in the underlying controls, as well as time lost on the driving task, resulting in our placing 18th in the DRC Finals.

Team TROOPER is composed of researchers from Lockheed Martin Advanced Technology Laboratories, the University of Pennsylvania (Professors Vijay Kumar and Kostas Daniilidis), and Rensselaer Polytechnic Institute (Professor Jeff Trinkle). This document details our system and performance in the DRC Finals in June 2015. Section 2 details our collaboration and design philosophy. Section 3 details the hardware, control, planning, and perception capabilities of the system. Section 4 discusses our performance in the Finals and Section 5 presents conclusions.

2 INTRODUCTION

The DARPA Robotics Challenge galvanized the robotics community to build and demonstrate field-ready robots that can be remotely operated to perform tasks in a disaster scenario unsafe for human presence. We believe that more important than winning the DRC competition was engaging with its community of researchers, and building the technology foundation for systems with the capability of accessing and altering complex human-engineered environments.

We believe that systems developed for the DRC provide the core capability necessary for continued research in advanced robotic platforms. These platforms are capable of operating in both human accessible, as well as otherwise inaccessible environments. They can also perform physical manipulation tasks that today require humans, and can one day replace the human in cases that would put them in harm's way. Our continued research in autonomy, intelligence, and perception will enable these robots to be reliably supervised by humans to achieve these missions.

2.1 Collaboration, FOSS, and Partnership

We recognize that the challenges of intelligent automation and control of a humanoid robot are larger than a single research organization can hope to achieve in the near future. For this reason, we have focused on synchronizing our efforts with the rest of the robotics community to maximize our impact. To do this, we identified and leveraged state of the art technologies to provide key capabilities, and we targeted our efforts on system integration and test as well as research for key capability gaps. This required a continuous assessment of community research, as several key developments were produced during the span of the DRC program.

To work effectively with the robotics community, we have adopted the open source Robot Operating System (ROS). ROS provides standards and practices that expedite the process of integration and testing of new software and hardware components. As a relatively small team, we heavily leveraged existing ROS components for the Virtual Robotics Challenge. As we proceeded on with the DRC Trials and later Finals, we decreased our reliance on stock ROS components and wrote our own. For example, we initially used the ROS SMACH state-machine framework before writing our own behavior managing framework. Additionally, we have wrapped existing ROS components in ways that add functionality, such as our adaptive perception manager, which provides easy methods to start, stop, and chain sequences of perception algorithms.

The DRC teams using the Boston Dynamics Atlas robot have embraced the spirit of cooperation by sharing solutions to hardware challenges. Several of the Atlas teams have provided system components as Free Open Source Software (FOSS), and we have made use of and contributed to those initiatives. We would like to see these collaborations continue to grow to enable our organizations to solve challenging research problems.

2.2 Simplicity and Autonomy

Fully autonomous humanoid robots were not feasible to develop for the DRC. However, a heavily teleoperated robot requiring significant operator expertise, training, and cognitive burden would not have been suitable for most operations. These robots would have been too slow, and the scenarios would not always have been rehearsable. To speed up operations of remotely deployed robots, increased automation is required. The human operator must be able to trust the robot with making most decisions without a human in the action loop.

To do this, we have designed a system that is meant to be autonomous, though with the ability to ask a human operator for guidance when necessary. This system design is intended to be platform agnostic; the philosophy and software components will later be applied to autonomous and unmanned platforms outside of humanoid robots.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

We begin by discussing our particular hardware solution, including both the DARPA-provided Atlas hardware and our own modifications. We then detail our software solution, covering algorithms and capabilities pertaining to low-level control, planning, and perception. Lastly, we detail which of the previous capabilities made their way into our competition-ready system and discuss our solutions on a task-by-task basis.

3.1 System Overview

The TROOPER system is designed to control the Atlas humanoid robot. Each arm of the robot can be mounted with a third-party end effector. Our left hand is equipped with a Robotiq 3-fingered adaptive gripper, and our right hand with a custom sensor hand. The head has a MultiSense SL stereo camera from Carnegie Robotics which includes a spinning Hokuyo LIDAR. This sensor package is mounted with an electric motor to tilt up and down. We integrated an additional VI-sensor from Skybotix to provide grayscale stereo vision to the left side of the robot for use in the driving task.

The TROOPER software runs on the Boston Dynamics Perception Box onboard the Atlas, which contains 3 Intel Quad Core i7-4700EQ processors. These computers are connected to the peripheral devices (such as the hands and MultiSense SL head) through Ethernet. A wireless router onboard Atlas connects the Perception Box to a remote operator control unit (OCU) with some packet loss. The OCU runs the user interface software. Some of the Atlas control software is replicated on the OCU in order to minimize data transferred over a low-bandwidth connection. The overall system diagram is shown in Figure 1.

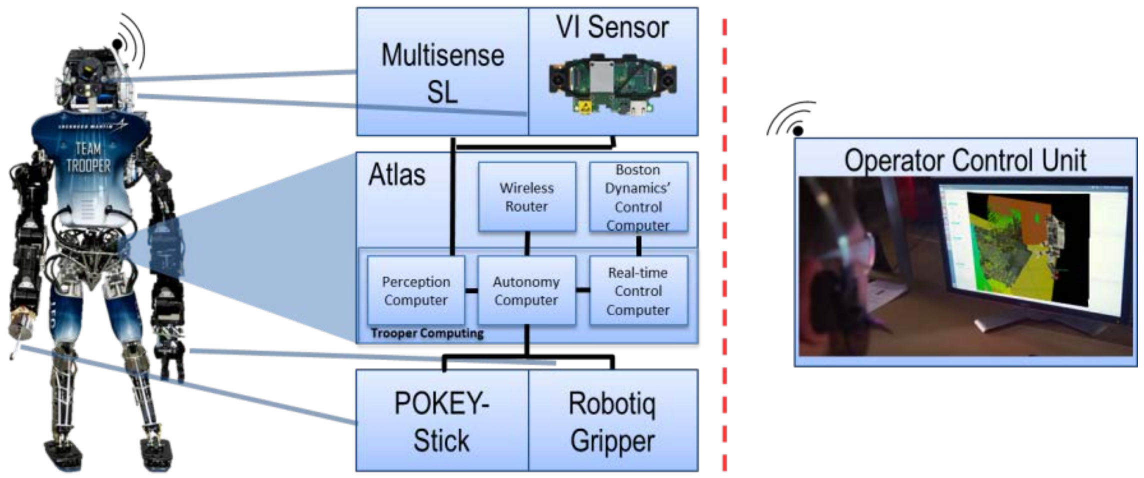


Figure 1. The TROOPER System for the DRC Finals

3.1.1 Robotic Hardware Components

Most disaster relief scenarios occur in human-engineered environments. Typical elements seen in a human environment include doorways, narrow hallways, stairs and ladders. Some of these elements are incorporated into specific tasks within the DARPA Robotics Challenge. In order to complete tasks within a human engineered environment, we decided to use a robot with a human form factor. The strength of this approach is apparent when performing tasks such as walking over a cinderblock pile. This task causes many pitfalls for wheeled robots, but a legged robot is able to walk over obstacles and place its feet at desired locations. Furthermore, the way in which the robot completes a given task can easily take inspiration from how a human would approach the same task. The Atlas robot, shown in Figure 2, satisfied the human form factor as described above. It also has the strength and dexterity to complete manipulation-based tasks such as turning a valve or cutting through a wall.

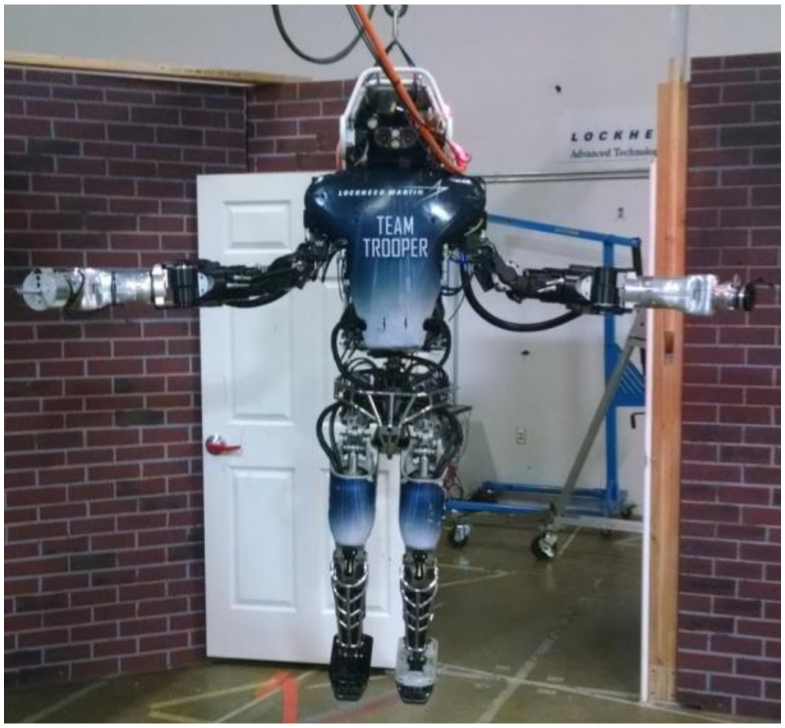


Figure 2. Atlas Robot

It was clear, however, that some modifications to the basic Atlas platform were needed to effectively compete in the DRC Finals. The robot itself comes with no standard set of hands. Instead, it was left up to individual teams to decide which set of manipulators they would use as end effectors. For the final configuration, the team elected to use a Robotiq 3-Finger Gripper on the left hand and a custom designed rod with accompanying sensors for the right hand. This configuration allowed for quick and robust solutions for the manipulation tasks.

3.1.1.1 Atlas Mechanical. The Atlas robot developed by Boston Dynamics is a bipedal humanoid robot that stands approximately 6 ft. 2 in. tall and weighs 390 lbs. The robot includes 30 actuated degrees of freedom. While most joints are hydraulically controlled, a select few are controlled with electric motors. The breakdown is shown in Table 1 and visually displayed in Figure 3.

Table 1. Atlas Joint Descriptions

Joint Name	Degrees of Freedom	Electric / Hydraulic
Neck	1	Electric
Back	3	Hydraulic
Right Leg	6	Hydraulic
Left Leg	6	Hydraulic
Right Upper Arm	4	Hydraulic
Left Upper Arm	4	Hydraulic
Right Lower Arm	3	Electric
Left Lower Arm	3	Electric

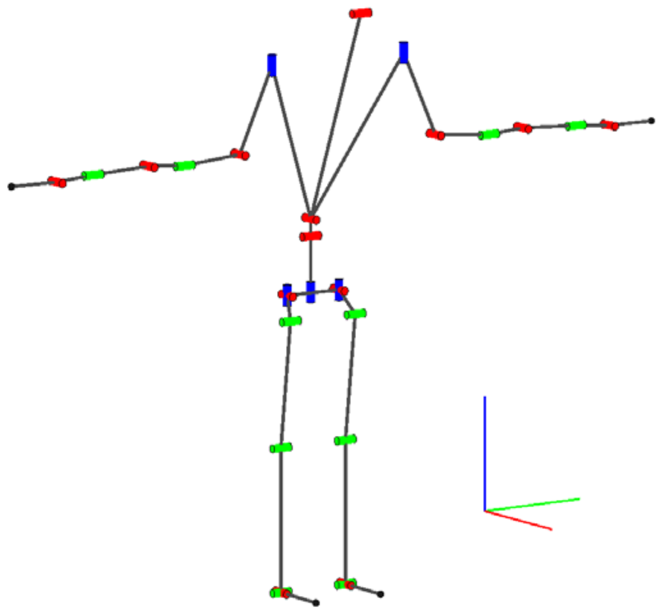


Figure 3. Atlas Joint Model

On each of the Atlas's wrists are 6-axis force/torque sensors as well as a 3-axis force/torque sensor in each of the robot's feet. Additionally, there is a 6-axis IMU that sits in the robot's pelvis cage. While testing before the Finals, the Atlas robot was equipped with a surrogate battery pack that would mimic the mass and center of mass of the actual battery pack. A battery emulator was connected to the robot via tether that would emulate the performance and profile of the battery. At the DRC Finals in June 2015, a 3.7kWh 165 VDC battery pack was attached to the robot to supply the robot with untethered power during each trial run.

For safety, the robot was equipped with 3 emergency stops (E-Stop), as shown in Figure 4. There exists one E-Stop on the back of Atlas, one wired E-Stop that would sit next to the OCU and a third wireless E-Stop that would be held by a testing assistant. These E-Stops could be engaged if the robot was ever in an unsafe or undesirable position.

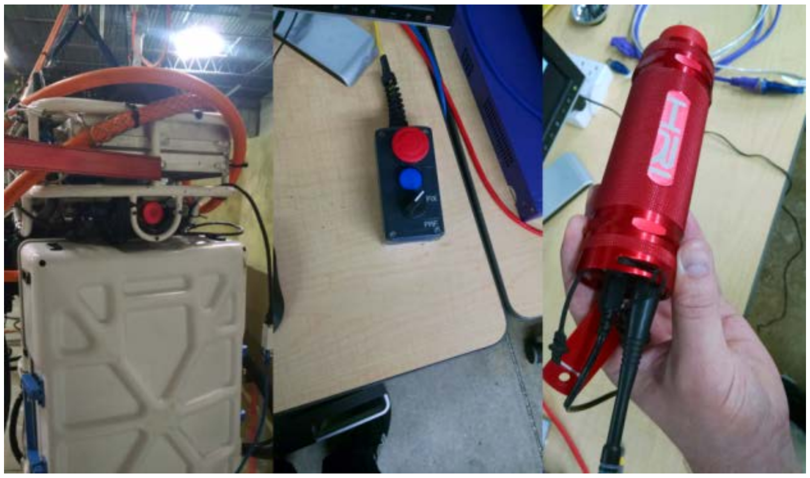


Figure 4. Robot Emergency Stops

3.1.1.2 Atlas Computing. The onboard computer system consists of three high performance computers and a closed hardware control box (provided and maintained by Boston Dynamics). No single computer was able to run the entire system due to the inherit complexity and number of peripheral components required. Each of the three computers runs a separate portion of the system (real-time whole body controller, perception computer, and autonomy computer). Accordingly, dedicated high speed links between each of the computers are used to communicate between components of the system. Each computer was also linked through a switch. While the outside connection was primarily used for communication from the robot to the user, it also allowed for a convenient way to debug and access each computer directly.

3.1.1.3 MultiSense SL. The main perception sensor onboard the Atlas robot is the MultiSense SL sensor head, shown in Figure 5. Developed and maintained by Carnegie Robotics, this sensor package is connected to the frame of the robot via a one degree-of-freedom electric joint in the neck. This allows the operator to tilt the sensor head up and down in order to get a better understanding of the robot's environment. There is unfortunately no pan degree-of-freedom; instead, the robot must yaw at its waist to look left or right.

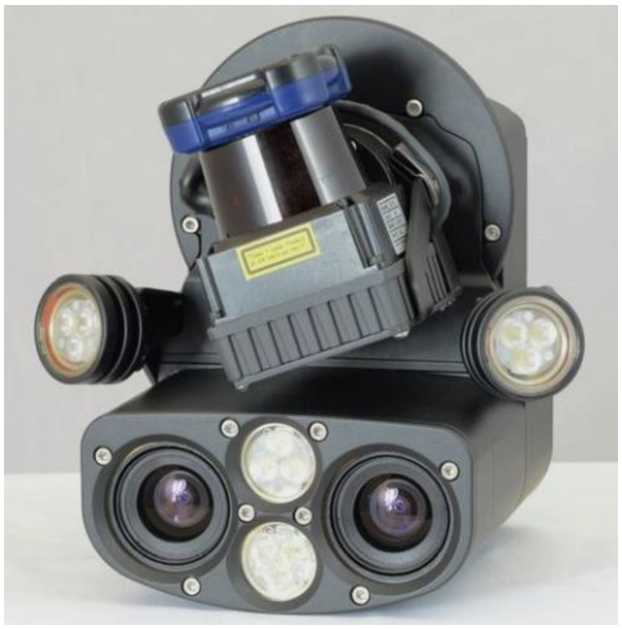


Figure 5. MultiSense SL

The sensor head includes a Hokuyo UTM-30LX-EW LIDAR on a rotating spindle to gather dense point clouds of the world. The MultiSense SL also outputs both stereo and monocular camera feeds. The real advantage of this sensor head is that most of the computationally intensive processes involved with collecting and processing perception data occur on an FPGA located inside the sensor. Specifically, the process of image rectification, stereo disparity mapping, and laser scan synchronization occur onboard, and are then output in an easy-to-use format. Furthermore, the MultiSense SL ships with a ROS-based API allowing for easier integration into the TROOPER system. The included API made it easy to not only receive perception data from the MultiSense SL, but also to change important onboard perception parameters. For instance, the operator was able to turn on the illumination lights on the MultiSense SL when working in a dark environment or change brightness and contrast settings when working in an extremely light environment. This flexibility allows the robot to get useful information from the camera in a variety of work environments.

3.1.1.4 Robotiq Manipulator. The Robotiq manipulator is a multi-finger underactuated industrial gripper. It provided an out-of-the-box solution to robustly grip arbitrary objects that fit within the hand itself. The gripper was also designed so that all of the contact surfaces are easily modified, which enabled a modification to turn on the rotary tool upon successfully grasping it. The two main downsides to this hand are its limited maximum aperture and its substantial weight. The small aperture is due to the limited range of motion of its fingers; the fingers in Figure 6 are shown fully open.

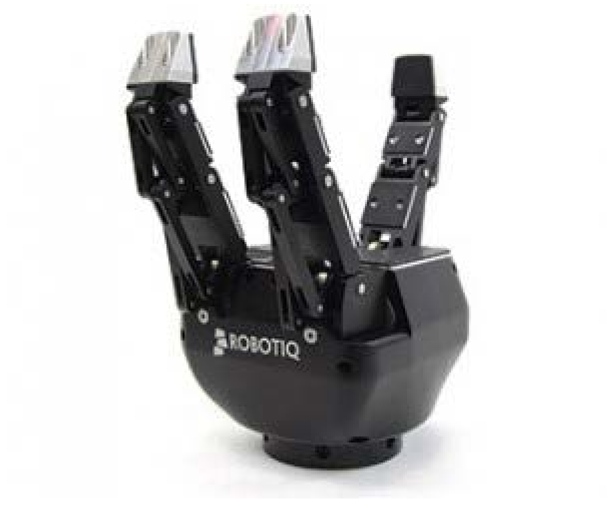


Figure 6: Robotiq 3-Finger Manipulator

We also found that when grasping an object that is not perfectly aligned in the palm of the gripper, the fingers will not exert the forces necessary to pull the object tightly into the center of the gripper. When manipulating the arm configuration, this can cause the object to shift in the hand and slip out of the grasp. While grasping objects, we command a cinching motion to the fingers that constantly relaxes and tightens the fingers at 1 Hz. This will enhance the grasp of an object when orientation shifts move it.

The Robotiq gripper proved to be useful in the driving, door, and wall task. During driving, this hand held onto the throttle mechanism (as described later) and was used to accelerate or decelerate the Polaris Ranger vehicle. In the door task, the gripper was put into a closed position allowing the end effector to push down on the door handle and unlatch it. In the wall task, the Robotiq gripper with 3D printed attachment was used to clench and activate the rotary tool.

3.1.1.5 POKEY Stick. The Pointed Object for Kinematic Extension without Yielding (POKEY) stick, shown in Figure 7, is a custom designed hand designed and built at Lockheed Martin ATL. This manipulation device provides both additional sensor information and capabilities to robustly complete several challenge tasks. The POKEY stick is equipped with an IDS UI-1005XS-C small form factor USB camera. The camera is positioned within the hand so that the center of camera frame is aligned with the stick. This design makes tasks involving visual servoing much easier to accomplish.

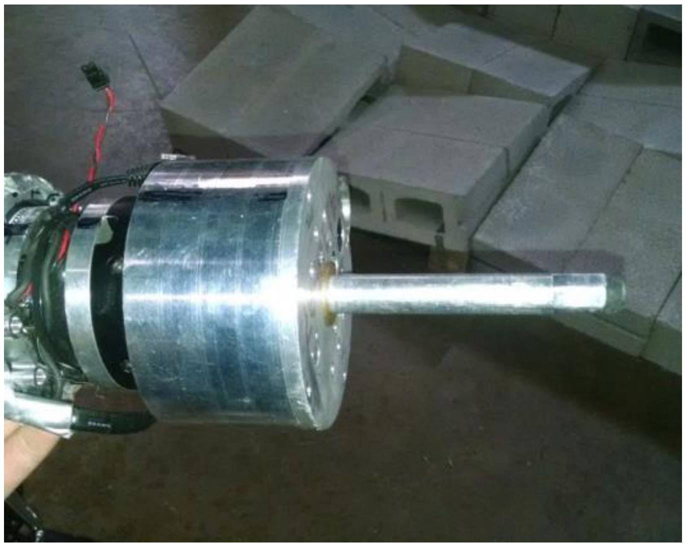


Figure 7. POKEY Stick

Since the camera is mounted at a more maneuverable position (i.e., end effector of the hand) than the MultiSense SL, the operator is able to use the hand camera stream to get perspectives of the environment not otherwise available. Using the camera in this way becomes very useful when performing manipulation work (for instance, when opening a door) that would otherwise occlude the view of the MultiSense SL. In addition to the available camera stream, the POKEY stick also includes a laser range finder to determine distance to an object, a force sensitive resistor to detect contact, and a microphone to determine if the drill has been activated. These peripheral sensors are connected to a Teensy microcontroller, which relays the sensor streams to one of the onboard Atlas computers via USB. The four main tasks in which the POKEY stick proved to be useful were the driving, valve, wall, and mystery tasks.

While driving, the POKEY stick allowed a connection to the steering mechanism with little slip. The setup allowed for the steering of the vehicle by only moving one joint in the wrist. More information about the driving mechanism is available in the following section. The POKEY stick was also valuable in the valve task; aligning one single rod in between the spokes of the valve was determined to be a quicker and more repeatable process than aligning a hand. By inserting the POKEY stick into the valve until the base of the hand touched the outer rim of the valve, the attached rod would remain in the valve throughout the turning motion. In the wall task, the cutting tool can be activated using a visual servoing approach to find the button with the hand camera then steadily move towards it with the rod until the force sensors and microphone indicate that the drill is on. Finally, the POKEY stick is useful in several of the mystery tasks including pulling an emergency shower chain or opening an electrical box.

3.1.1.6 VI Sensor. Skybotix's VI Sensor, shown in Figure 8, is a stereo camera system with an onboard FPGA currently used to stream both cameras and onboard sensors over a common interface. In the future, the FPGA will be used to calculate a disparity map and other vision elements to alleviate the load on a CPU. The sensor was oriented to look out the left of the robot, primarily for driving. This hardware addition is used to alleviate the fact that the MultiSense SL cannot pan left and right. Since the driving position of the robot is perpendicular to the direction of motion of the vehicle (to fit Atlas on our deployment slide), it was determined to be necessary to add another perception sensor to the Atlas in order to get a forward view of its environment. The slim profile of the VI allows for easy mounting within the protective cage that surrounds the MultiSense SL.

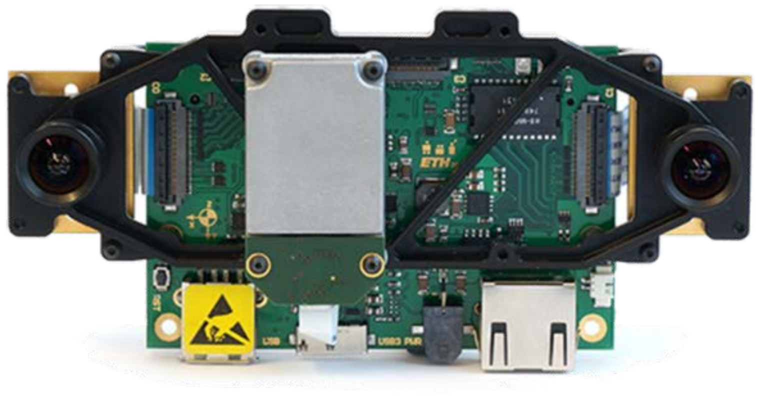


Figure 8. VI Sensor

3.1.2 Software. Robotic systems have great potential to assist humans in unsafe environments such as natural or man-made disaster sites. Their utility has already been demonstrated as rescue robots (Murphy, Kravitz, Peligren, Milward, & Stanway, 2008) and as bomb disposal robots (Carey, Kurz, Matte, Perrault, & Padir, 2012). However, these systems all relied heavily on a human operator to manually control a robot. Recent advances in sensing and autonomy have allowed for semi-autonomous systems (Chaomin, Yang, Krishnan, & Paulik, 2014)(Zhang, Lee, Yang, & Mylonas, 2014), relieving some of the cognitive burden on the human operator. Robotic systems will undoubtedly continue to become less reliant on teleoperation; however, due to the unstructured nature of real-world environments and the complexity of the required tasks, it is impractical to expect fully autonomous systems in the foreseeable future. The desired amount of human intervention and guidance is highly dependent upon the situation and will likely change throughout an operation. This has prompted research into the area of *sliding levels of autonomy* (Desai, Ostrowski, & Kumar, 1998) (Goodrich & Schultz, 2007) whereby a human operator may have varying levels of influence on a robotic system. Another related field of research is mixed-initiative interaction (Cacace, Finzi, & Lippiello, 2014) (Lomas de Brun, et al., 2008) in which the human and the robot collaboratively achieve goals by leveraging each other's strengths.

The TROOPER software system implements a paradigm that we refer to as *human-guided autonomy*. Human-guided autonomy incorporates ideas from both *sliding levels of autonomy* and *mixed-initiative interactions*. It further extends these concepts by incorporating a notion of *confidence,* which allows the robot to intelligently reason over multiple potential ways to achieve a goal, and to realize when it is appropriate to request human intervention. Every component in the TROOPER software framework, shown in Figure 9, was designed to support human-guided autonomy. All of the operator control unit components ran on a single desktop computer while the robot control unit components were split between two of the Atlas's onboard computers. We wrote all portions of the human-guided autonomy framework as part of the DARPA Robotics Challenge.

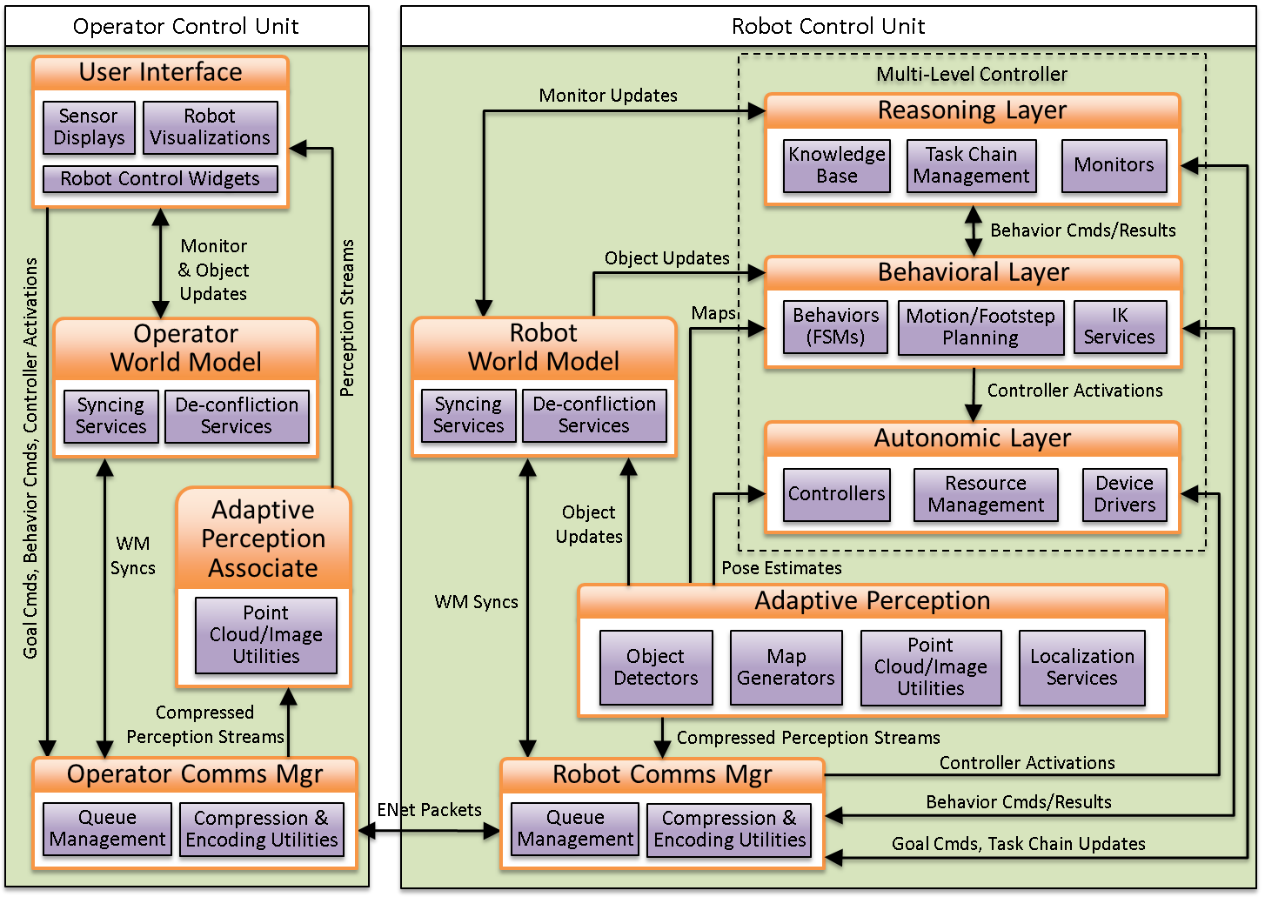


Figure 9. Software Architecture Diagram

3.1.2.1 Multi-Level Controller. The multi-level controller incorporates low-level controllers, high-level behaviors, and goal-based reasoning into a hierarchical framework that promotes re-usability and allows for operator intervention at all levels of the hierarchy. There are three distinct layers as shown in the overall software diagram above (Figure 9). Figure 10 shows some of the components from each layer that may be involved in commanding the robot to "grasp a drill". The interconnectedness of the layers can be seen as well as some of the knowledge base rules that would have been used to automatically generate the given task chain. Knowledge base rules and monitors are used in the reasoning layer to generate a sequence of tasks which dictate the parameters that are passed to the behaviors and controllers in the behavioral and autonomic layers.

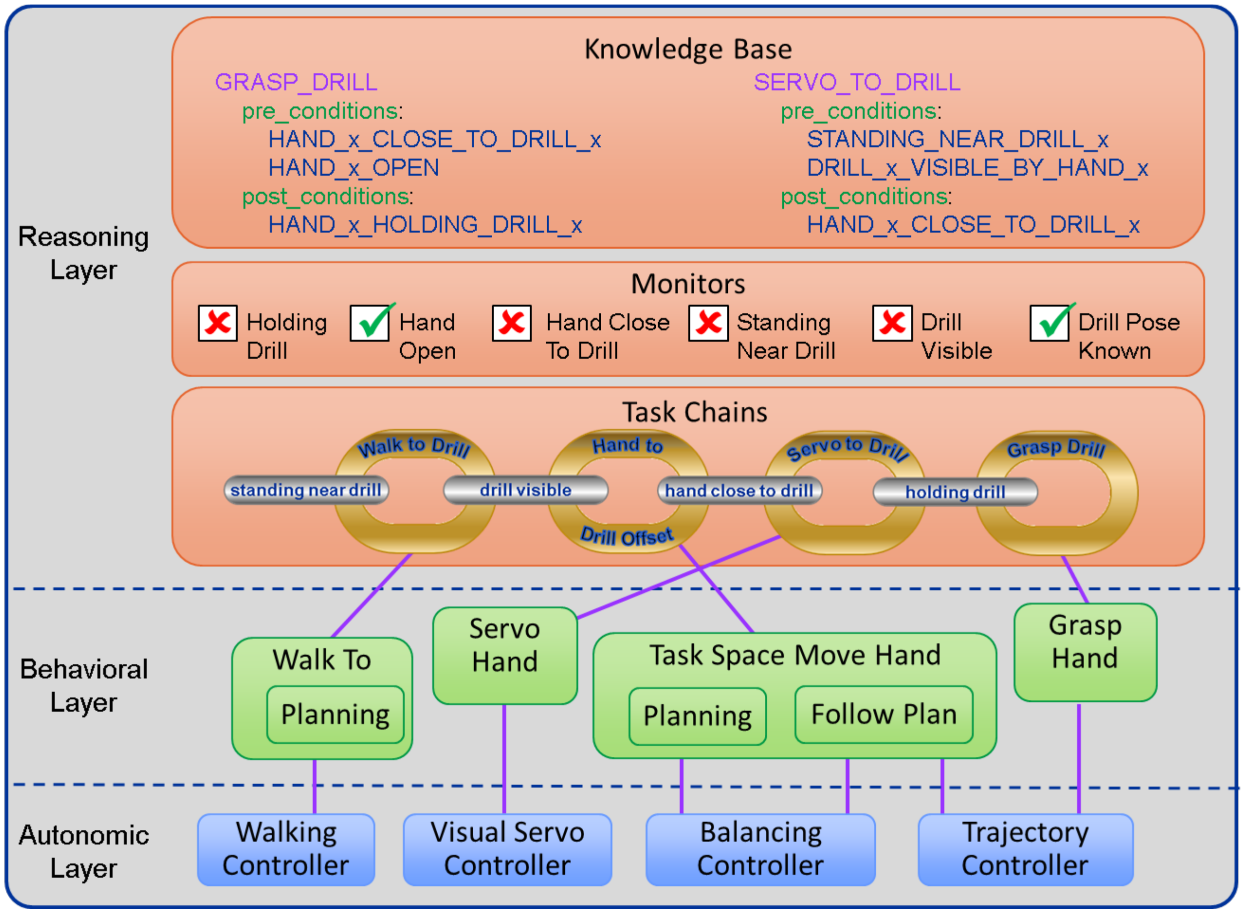


Figure 10. Multi-Level Controller Layers

Each of these layers is extensible using plug-in components. Multi-level controller plug-ins are only visible to the layers that are above them in the hierarchy. Most interactions with the multi-level controller are through the reasoning layer; however, the operator has the ability to interact directly with plug-ins in every layer.

3.1.2.1.1 Autonomic Layer. The lowest layer of the multi-level controller contains plug-ins that deal most directly with the hardware, or simulated hardware. Three different types of plug-ins can be found at this layer: *hardware components*, *controllers*, and *real-time services*.

Hardware Components. Most robots consist of multiple pieces of hardware, each with its own control interface. For instance, the Atlas robot has a unique control interface for the MultiSense SL head sensor as well as each different type of hand. Every individual type of hardware that has a unique control interface is represented by a corresponding *hardware component* in our system. A hardware component is responsible for sending commands to the hardware and receiving observed data from the hardware. On startup, each component registers itself with a generic *robot* object which makes the individual hardware components available to the rest of the system. This modular design lets us rapidly switch between various hardware configurations and allows the rest of the system to remain mostly platform agnostic. We created the following hardware component plug-ins for the DRC:

- BDI Atlas
- IHMC Atlas

- MultiSense SL head
- iRobot hand
- Robotiq hand
- Sandia hand
- SRI hand

Controllers. Controllers are responsible for generating joint-level commands and passing them to the appropriate hardware components. Since some controllers may be designed for a specific hardware component, each controller has the ability to specify one or more required hardware components. If these hardware components are not available on startup, the controller will not be initialized. Multiple controllers may be active at any given time so to avoid conflicts, a controller is required to reserve a joint before commanding it. When multiple controllers attempt to simultaneously reserve the same joint, a priority scheme is used to determine which controller is allowed to take ownership of the joint. We used the following controllers during the DRC:

- Balancing
- Boston Dynamics Proxies (locomotion, manipulation, stand prep)
- Cartesian Velocity
- Hand Cinching
- IHMC Proxies (locomotion, balancing)
- Maintain Current Joint Position
- Execute Joint Trajectory

Real-time Services. Many of the controllers in the autonomic layer have overlapping requirements, such as filtered IMU and force torque data from the robot. Additionally, the autonomic layer is responsible for providing information such as joint states and robot pose to the rest of the system. These tasks are accomplished using plug-in services. The services provide a simple method of broadcasting information at rates other than that of the autonomic layer's main control loop. These services also allow for the consolidation of any computations that are required by multiple controllers. We used the following services during the DRC:

- IMU Filtering
- Force Torque Filtering
- Joint State Publishing
- TF (Transform) Broadcasting
- Robot Pose Publishing
- Robot State Encoding

3.1.2.1.2 Behavioral Layer. The behavioral layer contains a collection of simple actions and perception routines, all of which are referred to as *behaviors*. We have implemented behaviors as hierarchical state machines using Boost Meta State Machine (Henry, 2009). Each behavior is parameterized, allowing it to be customized for the particular task at hand. For instance, in addition to a 2D goal, the *Walk To* behavior accepts parameters such as swing height, foot spacing, and step distance. Behaviors utilize controllers from the autonomic layer as well as perception streams from the adaptive perception manager. Multiple behaviors can be executed in parallel, assuming that the controllers they utilize do not conflict with each other.

Components in the behavioral layer often require information about the current state of the robot. This information is made available through an object called the *Robot Model*. The Robot Model is a kinematic representation of the robot that is updated from joint state messages that are published from the autonomic layer. Components can use the Robot Model to monitor the current state of the robot as well as to plan for feasible robot configurations. The Robot Model provides inverse kinematics utilities, access to the robot's pose in the world, and convenience methods for determining its center-of-mass when in a given configuration.

3.1.2.1.3 Reasoning Layer. The reasoning layer utilizes behaviors from the previous layer to achieve complex goals. A *behavior* can be thought of somewhat like a re-usable template. A specific instantiation of behavior, whose parameters have been specified to achieve a particular goal, is something that we refer to as a *task*. To achieve a complex goal, one that itself contains multiple sub-goals, several of these tasks must execute in sequence. We refer to these sequences as *task chains*.

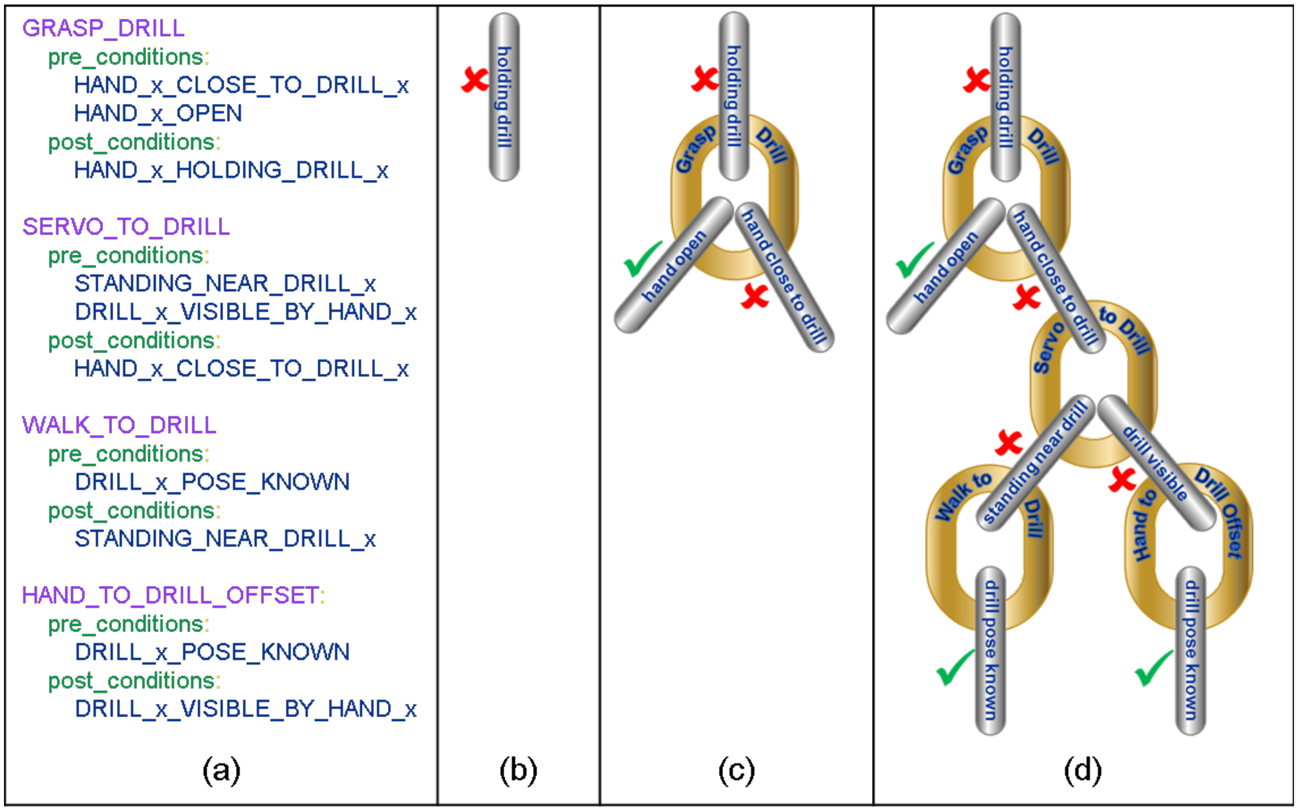
Several components are involved in building and executing task chains. These include: A *knowledge base* with information that allows a task chain to be automatically generated for a given goal; *monitors* which continuously check for conditions that indicate that goals, or sub-goals, have been satisfied; a *reasoner*, which interacts with the user interface to generate and modify task chains, and then interacts with the behavioral level to manage tasks during execution.

The knowledge base, which is defined in a configuration file, contains *rules*, which describe the pre-conditions and post-conditions of all tasks. For instance, the *detect drill* task has a pre-condition that the "robot is in a pose that allows it to see drill x", and after executing this task, the post-condition will be that "drill x's pose is known". Multiple pre-conditions can be defined for a task but the knowledge base currently only supports a single post-condition for each task. These pre-conditions and post-conditions eventually become goals and sub-goals in a task chain. The knowledge base also contains information that maps variables in pre-conditions to corresponding variables in post-conditions. In the previous example, there is only a single variable, "x", which identifies a specific drill, but many rules contain multiple variables. These variable mappings are used to automatically propagate behavior parameter values from the top-level goal provided by the operator to all of the sub-goals in a task chain.

Every goal and sub-goal in a task chain has an associated monitor. Monitors are created when a task chain is being built and they persist for the lifetime of the task chain. There are multiple flavors of monitors but, once created, monitors continuously evaluate their conditions. This allows a task chain to be opportunistically modified if a sub-goal becomes satisfied earlier than expected. Conversely, if a pre-condition which had previously been satisfied becomes unsatisfied during execution, additional sub-goals can be added to the task chain to deal with this contingency.

The reasoner is the main executive of the reasoning layer. It builds task chains and then manages them throughout their lifetimes. Task chains are actually composed of a combination of *goal links* and *task links*. A goal link represents a pre/post-condition from the knowledge base and a task link represents the task that will be executed to achieve the associated goal.
An operator has the ability to create a custom task chain by manually specifying all of its steps, in which case the reasoner simply handles execution of the task chain. Alternatively, the operator can send the reasoner a high-level goal, and the steps in the task chain will be automatically inferred by the reasoner. In either case, the operator always has the ability to modify a task chain once it is created. Task chains are automatically inferred using a process similar to backward-chaining (Russell & Norvig, 2003, pp. 337-344). Given a high-level goal, the knowledge base is queried for all rules whose post-condition matches the goal. Monitors are created and evaluated for the pre-conditions in these rules, and if any of the pre-conditions are

unsatisfied, the process is repeated with the pre-condition as a sub-goal. This continues until a rule is reached whose pre-conditions are all satisfied. Figure 11 shows an example of how this process may work to generate a task chain for picking up a drill. Label (a) shows a sample knowledge base. In (b) a goal link and monitor is created for the given goal of “holding drill”. In (c), because the monitor is unsatisfied, a matching post-condition is found in the knowledge base and the corresponding task is added to the chain along with goals and monitors for the corresponding pre-conditions. Lastly, (d) shows the process is repeated until every sub-goal is either satisfied or has a child task which will be used to satisfy it.



```
GRASP_DRILL
   pre_conditions:
      HAND_x_CLOSE_TO_DRILL_x
      HAND_x_OPEN
   post_conditions:
      HAND_x_HOLDING_DRILL_x

SERVO_TO_DRILL
   pre_conditions:
      STANDING_NEAR_DRILL_x
      DRILL_x_VISIBLE_BY_HAND_x
   post_conditions:
      HAND_x_CLOSE_TO_DRILL_x

WALK_TO_DRILL
   pre_conditions:
      DRILL_x_POSE_KNOWN
   post_conditions:
      STANDING_NEAR_DRILL_x

HAND_TO_DRILL_OFFSET:
   pre_conditions:
      DRILL_x_POSE_KNOWN
   post_conditions:
      DRILL_x_VISIBLE_BY_HAND_x
```

Figure 11. Reasoner Task Chain

When multiple rules are found with the same post-condition, they are both added to the task chain as children of the same goal link. This fork in the chain represents a decision point, with both sub-chains capable of achieving the same goal. Once the entire task chain has been built, all of these decision points are evaluated and the optimal path through the chain is presented to the operator for approval. The optimal path is determined by calculating a confidence value for each goal link and then, at each decision point, selecting the sub-chain that results in the highest confidence that the overall goal will be achieved. Confidence for a particular goal link is calculated using Equation (1). Where $C_{monitor}$ is the confidence that the goal condition has actually been satisfied, $C_{sub\text{-}goals}$ represent the confidence values of all pre-conditions of the given goal, and C_{rule} is a pre-determined measure of the confidence that the goal can be achieved if all of the pre-conditions have been met. The rule confidence is currently specified in the knowledge base, but future research should enable this value to be learned over time.

$$C_{goal} = \begin{cases} C_{monitor} & \text{if goal monitor satisfied} \\ \min(C_{sub-goals}) * C_{rule} & \text{if goal monitor unsatified} \end{cases} \quad (1)$$

During execution of a task chain, as sub-goals are completed and monitors become satisfied, goal link confidence values are updated. If the updated confidence values result in one of the currently selected sub-chains becoming sub optimal, the reasoner is able to dynamically modify the task chain. The reasoner also uses these updated confidence values to determine if it is safe to proceed with the next task in the sequence or if confidence has dropped too low and the operator should be asked to intervene.

Interventions can also be initiated by the operator. For instance, if the robot is attempting to unlatch the door and the operator notices that the robot is not quite aligned with the door handle, the operator can pause the task chain, and then either modify existing task parameters or manually teleoperate the robot's hand into a new start position before unpausing the task chain and allowing the robot to continue. Another way that the operator collaborates with the robot is by updating objects that exist in the world model. Many of the behaviors that the robot performs are with respect to some object in the world. For instance, the goal for a *walk to* behavior may be relative to a door that has been identified in the shared world model. The operator can influence this behavior by simply updating the pose of the door in the shared world model. This update will automatically trigger an update to the parameters of all relevant behaviors in an existing task chain.

3.1.2.2 Adaptive Perception. Traditionally autonomous systems follow an Observe-Orient-Decide-Act (OODA) model, shown in Figure 12. This paradigm involves receiving information about the state of the world from raw sensors then continuously processing the data to determine how to effectively control actuators to modify the world and transition into a new state. Through continuous perception and action, the system converges towards reaching and observing its goal state.

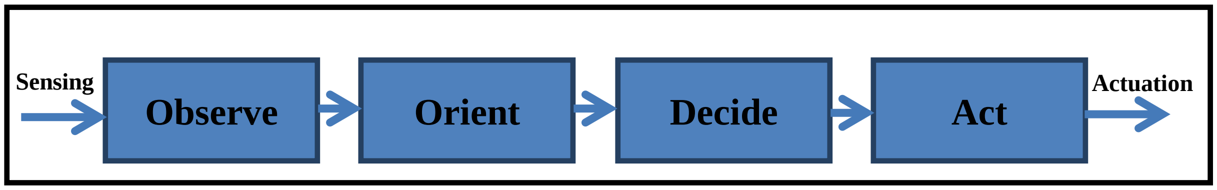


Figure 12. The Observe-Orient-Decide-Act (OODA) Loop Decision Cycle

This paradigm has worked well for autonomous systems with a small responsibility set, where the number of goals is relatively small (e.g. less than 50). From a perception standpoint, the autonomous system needs adequate sensing to correctly recognize the world state to determine progress towards and completion of a goal. For a small number of goals, it is very common for perception to be achieved through a single sensing modality (e.g. vision) with a high degree of overlap and reuse of algorithms that can be engineered into the system's processing. It is also common to use highly specialized sensors designed for observation of specific events, eliminating the need for complex processing by directly observing the quantity of interest. However, multi-mission autonomous systems that are capable of a diverse set of actions and goals will have perception needs with varying degrees of overlap, and redundantly processing all of the tasks simultaneously becomes computationally intractable with onboard computing resources.

We have observed in nature that sensing is context-driven. For example, when walking on the streets of a crowded city, a person will assume the ground is flat pavement and can place footsteps open-loop, while focusing their attention of observing the motion of other people and predicting their trajectories to avoid collision. In another context, a person hiking through the woods does not need to predict motion of other actors, but needs to focus attention on classification of terrain and precise placement of footfalls. We believe that autonomous systems need to similarly be **context-driven** in deciding how to process sensor data (as shown in Figure 13), and **adaptive** to identify and exploit redundancy in concurrent processing. We also wrote the adaptive perception manager as part of the TROOPER effort. The underlying processing modules utilize functionality found in ROS, but the perception manager is a new creation.

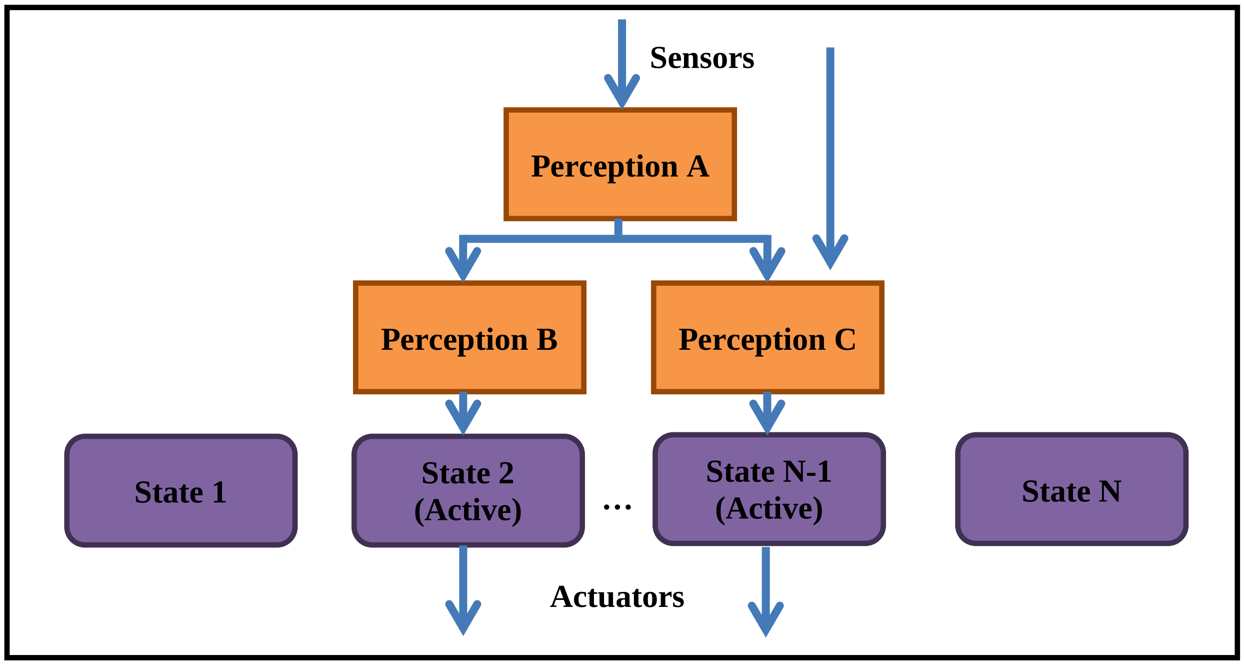


Figure 13. Context-Driven Observation Model

Our adaptive perception system contains a centralized perception server responsible for managing sensor processing to generate information streams at the request of perception clients. The perception clients can request and subscribe to data streams. They are provided with a mechanism for describing the processing required to generate their data, including the sensor source(s), algorithms, and parameters. The perception server dynamically generates the necessary processing chain, reusing existing computation when possible, and publishes the resulting data for the clients to consume. The client-server mechanism is illustrated in Figure 14.

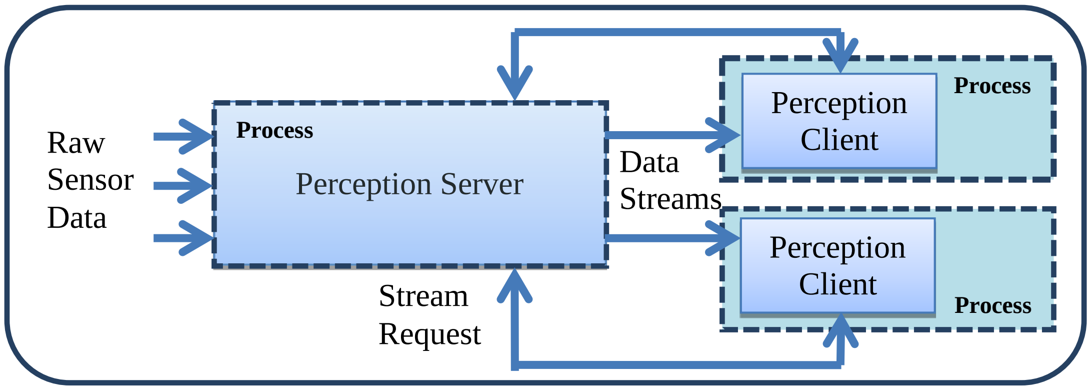


Figure 14. Adaptive Perception Client-Server Mechanism

A suitable analogy to the perception system is that of a restaurant. Customers at a restaurant are given a menu of items that can be served, along with a finite set of parameters that can be adjusted (e.g. substitute French Fries for a side salad). The kitchen staff is responsible for servicing these requests, and they optimize their kitchen workflow based on the current set of orders. They allocate their time and resources across multiple items, and duplicate tasks when possible. For example, French Fries will likely be replicated across multiple orders.

The perception server is responsible for executing all sensor data processing for a system. On startup, the server initially executes only a 1 Hz spin cycle to check for incoming stream requests and publish server status. When a perception client requests a stream, the request describes the input stream name(s), modules to process on the stream(s) along with their parameters, and the desired rate of the resulting data. If needed, the server will subscribe to the data stream from the sensor, instantiate processing modules with the requested parameters, and publish the requested resulting data. An example flow can be seen in Figure 15. The perception client requests Input 1 processed by Module 2 and Input 2 processed by Module 1. The perception server creates the subscribers, processing modules, and publishers if they do not already exist, otherwise it will use the existing streams. From the request, the Server also knows that Module 2 requires both Input 1 and Input 2 to function.

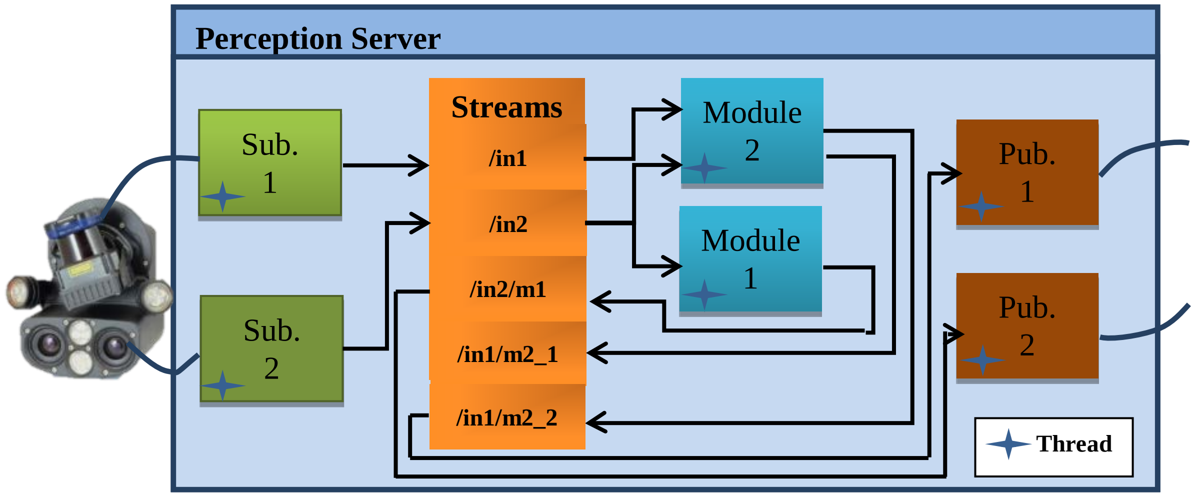


Figure 15. Example Processing Flow inside the Perception Server

Internally, our perception server executes processing modules in separate threads. These threads pass data between one another through streams. Streams contain perception data in raw or processed forms along with meta-data. Because all of the modules share the same process, their threads can share the data in their streams by maintaining locks rather than copying memory. This also avoids the need to serialize data for intra-process communication, which is generally an expensive operation for point clouds and camera images. Streams maintain count of the number of modules and publishers that are listening, and modules will automatically deconstruct their threads when all output streams have no listeners.

The timing of threads in the perception server maintains regular synchronization despite variability in processing time through use of condition variables. Condition variables are objects that block the calling thread until notified to resume. When a module or publisher listens to a stream, it passes its condition variable to a list maintained by the stream. When a subscriber or module updates the data in a stream, it notifies the condition variable of all listeners. This guarantees that the thread of every listener will become active when new input data is available. However, when woken up early, our threads will choose to go back to sleep for the remainder of their cycle time to maintain their desired rate. The alternative would be to blindly spin at the desired rate, potentially processing old data on late arrivals. This event-based throttling framework ensures regularity in data processing rates in complex multi-threaded processing chains. Shown in Figure 16, the timing system is throttled to maintain steady timing across arbitrarily large multi-threaded processing chains with low latency.

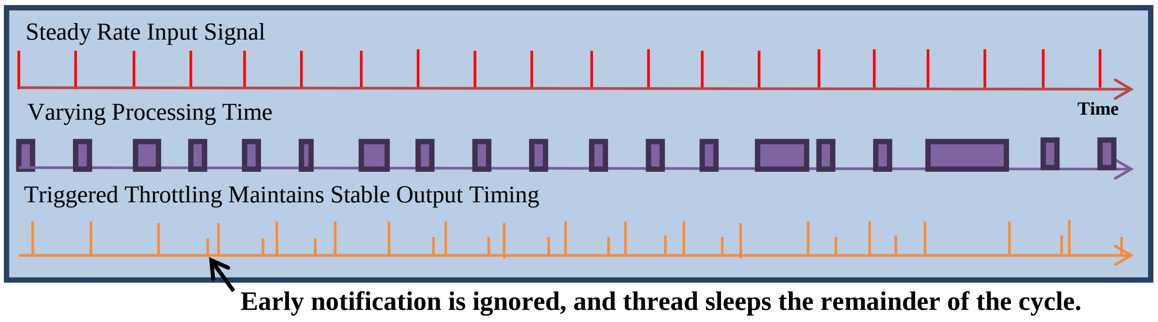

Steady Rate Input Signal
Time
Varying Processing Time
Triggered Throttling Maintains Stable Output Timing
Early notification is ignored, and thread sleeps the remainder of the cycle.

Figure 16. Event-Based Timing System

For distributed systems, rather than a single perception server, we expect to develop a mechanism for multiple servers to coordinate their processing. Each server may have some subset of sensors available locally, while others can be accessed by requesting streams from another server. In bandwidth restrictive networks, they will need to use compression modules and other techniques to minimize the size of the stream that is sent through the network. There is also the possibility that these servers are separated from an end user by a restrictive network. For this case, we developed a perception associate as shown in Figure 17. This is a light-weight variant of the perception server that is useful for building streams on behalf of a human operator.

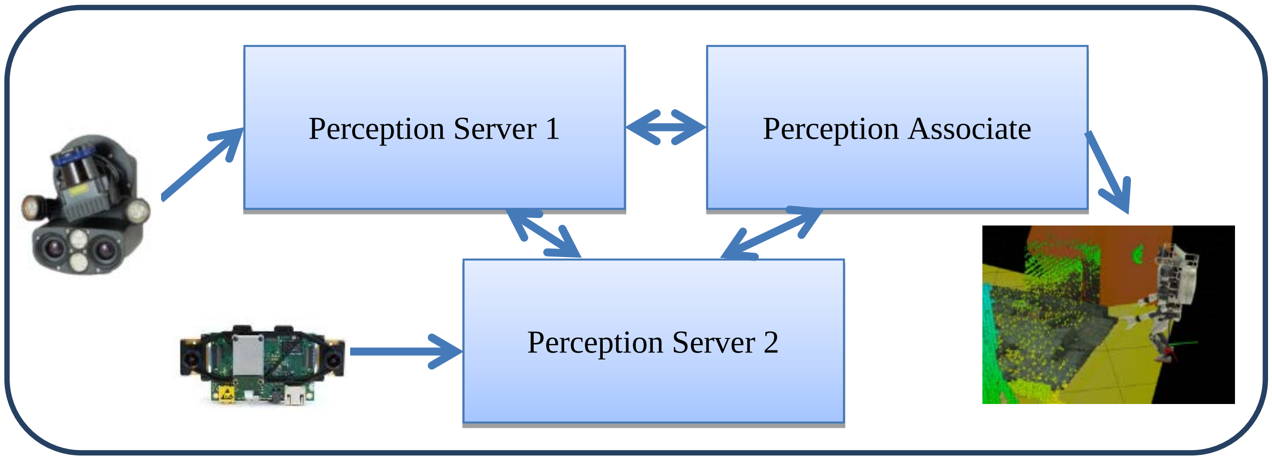


Figure 17. Framework Scales across Distributed Sensing and Computing Architectures

Importantly, the adaptive perception system contains a large number of perception modules, ready to process any type of sensor data produced within the TROOPER system and be chained as requested by a perception client. The available perception modules are collected in Table 2 below.

Table 2. List of Perception Modules Developed for DRC, Sorted by Category

Point Cloud Utilities	Robot Self Filter	Point Cloud Coloring	ROI Crop	Scan Assembler	Downsampler
Localization	Point Cloud Registration	Visual Odometry			
Mapping	OctoMap	Traversability	Height Map		
Segmentation	Plane Fitting	Clustering			
Tracking	JPL Fiducials	Alvar Markers			
Detection	Door	Valve	Cutting Tool	Tool Button	Terrain Field
Compression	Octree	H.264	JPEG	Image Patches	Point Cloud Sampling

3.1.2.3 Communications. DARPA specified the allowed communications between the robot, optional field computers, and operator stations as shown in Figure 18. The Atlas robot maintained bidirectional 300Mbps wireless communications with an access point. At this point, an optional field computer could be placed; we chose instead to keep all of our robot-side computation onboard the Atlas platform. Two logical links were provided between the robot control unit (RCU) and operator control unit (OCU) networks, keyed by UDP port number. The first logical link is intended for low bandwidth telemetry and robot control data and is a bi-directional connection limited to 9600 bps. The second logical link is intended for high bandwidth sensory data from the RCU to the OCU unidirectionally at 300 Mbps.

Beginning a run, and until the robot had passed through the door to the simulated indoor environment, link 3 functions without blackout. Once the robot is through the door, the unidirectional link operates in isolated, intermittent one second bursts. Blackouts between bursts vary in length, but begin at approximately 30 seconds in length and decrease in time as the run continues. At 45 minutes into a run, the blackouts decrease to zero and the unidirectional link functions unimpeded.

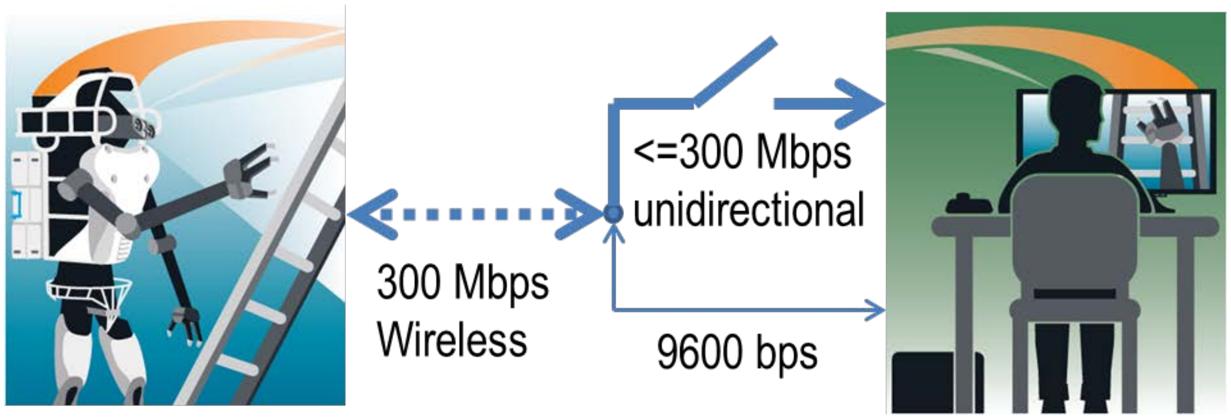


Figure 18. DRC Finals Communications Setup

ROS provides a communications API designed to deliver information between well connected components of a robot. However, the underlying network protocol is not well suited for the poor connectivity between RCU and OCU within the DARPA specified communications setup. In order to allow ROS messaging to serve the whole TROOPER system, we split ROS cores between RCU and OCU and then bridge the gap with the TROOPER communications manager. This network protocol bridge uses a combination of reliable and unreliable protocols on top of UDP to connect ROS messaging between the RCU and OCU. The TROOPER communications manager adds robustness to the connection of the links between RCU and OCU allowing reconnection without interruption should either side need to be restarted. The communications manager maintains channels that correspond to ROS topics and the fair queuing of messages from different topics.

In addition to link level control, the parameters of the DRC Finals resulted in the need for heavy compression and conservation of bits. The control data is compressed to minimize bandwidth consumption. For example, joint position data from the robot is represented by 8-bits per joint, providing nearly 1 degree resolution for most joints. We also use deterministic planning mechanisms so that both the operator and robot can generate the same motion plan, and only the start, goal, and plan label need to be communicated.

Our perception data is separated into pieces that individually contain all necessary meta-data with less than our 1440 byte payload MTU size. This allows data to reach the operator even in the presence of severe packet loss. For example, our camera images are separated into patches that are at most 20 by 20 pixels, and each of these patches contains meta-data for the size of the complete image, as well as the location of the patch relative to the image. Thus, even if the operator does not receive all of the data packets, they are able to view the parts of the image that were received. We randomize the order of transmission to minimize the spatial correlation between lost packets. The resulting image, despite heavy packet loss, is shown below in Figure 19.

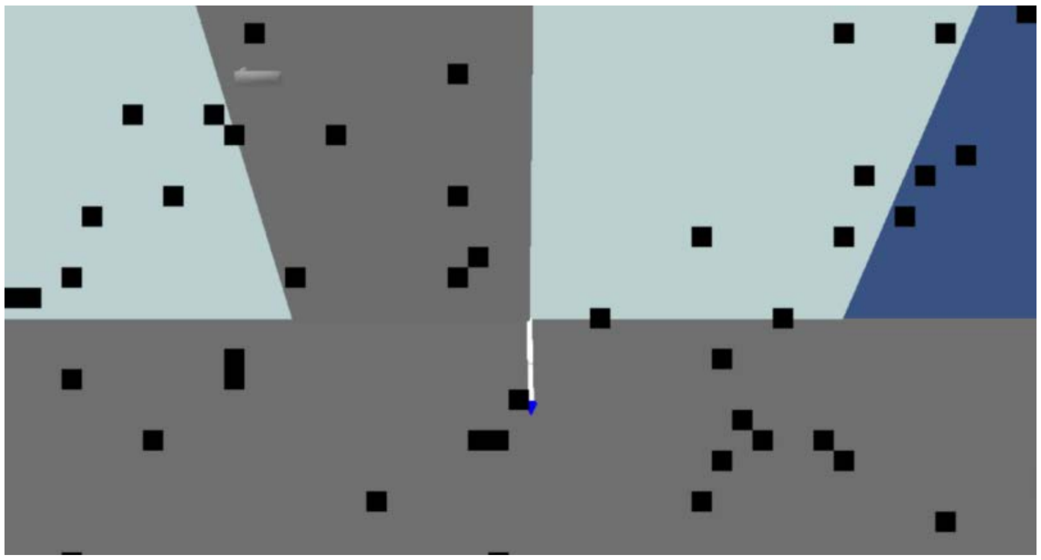


Figure 19. Image Blocks

3.1.2.4 Shared World Model. Controlling a robot using task-level commands necessitates a symbolic language that describes the robot's environment and can be used by the human to identify objects that the robot should interact with. This allows the human operator to specify a task such as "pick up the drill" and the robot to understand what a drill is and where it exists in the world. For our purposes, this symbolic language took the form of ROS messages. The role of the shared world model is to facilitate a process in which the robot and the human collaboratively refine a shared view of the robot's environment.

A robot with perfect perception capabilities could simply tell the human operator what is in its world. However, this is not possible with current technology. Conversely, it is not desirable to place the burden of interpreting sensor data solely on the human. The shared world model allows for collaboration between the two. For instance, it is possible for the robot to provide an initial estimation of an object's pose and for the operator to simply refine that estimate. The shared world model is essentially a distributed data store whose key feature is a mechanism for resolving conflicts between the robot's view of the world and the human's view of the world. The TROOPER system contained two instances of the shared world model: one on the operator control unit and one on the robot. Each instance consists of multiple layers as shown in Figure 20.

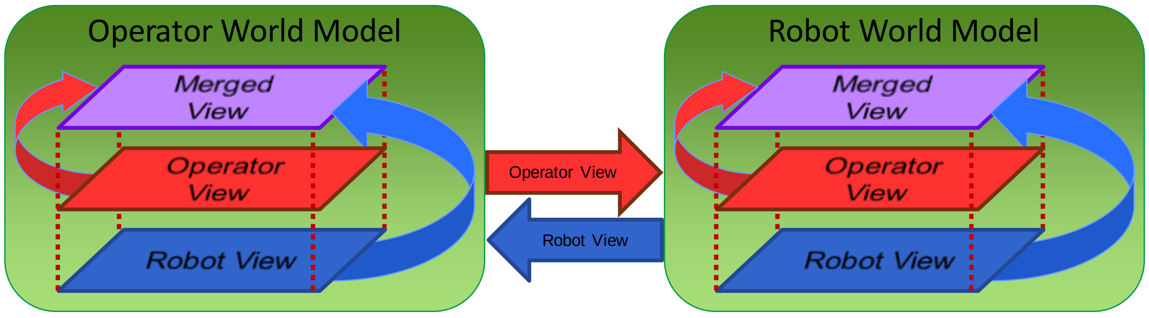


Figure 20. The World Model Sync Process

Changes to the robot's view are added to the robot world model while changes to the operator's view are added to the operator world model. These views are then shared with the other world model instance through a syncing process. After syncing, both instances will contain identical operator and robot views. Whenever another system component, such as planning or collision detection, requests information about an object in the shared world model, the operator and robot views are merged together to provide the shared view of that object. In the simplest case, the merging process is simply a union of the two views. When there is a conflict between the operator and robot view for a particular object, however, one of several merging strategies must be employed. Currently these strategies include:

- Priority is given to whichever layer is currently specified as the "authority" for a given object.
- Priority is given to the most recent update.

The currently designated "authority" for a particular object can be specified at any time. For instance, if an operator modifies the orientation of an object that was detected by the robot, the authority would be set to human. However, if the perception system then begins to track the object, the robot would once again become the authority.

The contents of the shared world model are not restricted to physical objects in the world. Any type of data that may be modified by both the human and the robot is a potential candidate for inclusion in the shared world model. At the time of the Robotics Challenge Finals, the shared world model contained objects (drills, doors, walls, etc...) as well as information about the slope of cinder blocks and the status of monitors that are used by our high-level reasoner to coordinate task sequences.

3.1.2.5 User Interface. As a practical interface, the UI offers direct windows to important parts of the multilevel controller, adaptive perception manager, and world model. It gives the user full input into the reasoner, enabling construction of tasks and task chains. It informs the user about the state of the controller manager: the running controllers and actively controlled joints. It renders the state of the world model and the published ROS topics of the adaptive perception manager: renderings of known objects, point clouds, and other 3D visualization data. These facets combine to allow the user to be an effective actor in a dynamic human-guided autonomy system. The UI is our own creation as part of the TROOPER effort, utilizing the Panda3D open source rendering engine and Qt libraries. In retrospect, we would have saved development effort by instead modifying the existing ROS visualization tools (rviz) rather than pursuing UI and rendering development from scratch.

The UI was designed to maximize essential concepts pertaining to the user experience:

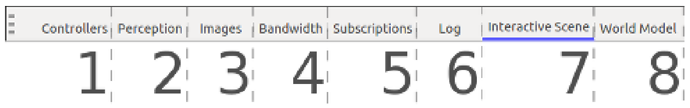
- Direct – The UI presents information as quickly and straight-forwardly as possible.
- Evolving – The UI is meant to be updated quickly as new features are added or modified.
- Simulating – The UI provides previews of motion plans and expected odometry whenever possible. The user is able to preview what the system believes will result from the commands it is given.
- Partitioned – The UI is removed from processor-intensive features whenever possible. Data is read from ROS topics that are generally available. Most dialogs send command information directly to a controller or manager on the robot.
- Analytical – The UI should provide as much data as possible in such a way that informs the user.

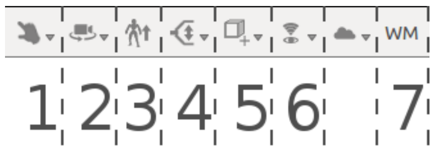
- Intelligent – The UI helps the user with more complex concepts such as task chains and the perception pipeline with graphical cues and presetting parameters based on contextual knowledge.
- Robust – The UI is not dependent on connection to the robot and vice-versa. Crashes on either side are not to affect the other.

These concepts drive toward the goal of streamlining the human aspect of human-guided autonomy. The user must have as much information as possible about what actions the robot believes it will undertake while still interfacing with an interface removed from the robot system.

The UI utilizes a tabbed main view, a side pane, and a full bar of quick actions. Tabs are comprehensive representations of a single aspect of the TROOPER system. For instance, the Interactive Scene tab shows the current kinesthetic and visual knowledge of the robot and has shortcuts to planning based upon this information. The Controllers tab shows the lower-level perspective of which control algorithms are running and on which actuators. Combined, these tabs could flood the user with information that is irrelevant to their current focus, so only one is shown at a time. The side pane lists active processes running on the robot: task chains/behaviors, controllers, and telemetry data being parsed. All of these subsections possess shortcuts to do important actions directly on these processes (e.g. stopping the MultiSense SL streaming data) or initialize new ones. The quick action bar focuses on making certain immediately needed functionality accessible in one or two clicks.

Typically, tabs are spread across two windows with the side panel and quick action bar duplicated on both. The secondary window contains tabs that focus on analysis and system information, i.e. the Bandwidth tab or the Controllers tab. The primary window contains tabs that focus on direct user interaction, the Interactive Scene and World Model tabs. This philosophy means that the user can glance at the secondary window to supervise the lower-level system while utilizing the primary window to direct the robot. No matter where the user has focused their attention, they can quickly access immediate concerns via the side pane and the quick action bar. The images in this document show the UI elements combined onto a single screen for easier reading. Figure 21 shows the combined UI window, with callouts for the tabs (A), side panel (B), and quick action bar (C). The annotated UI elements are also gathered in Table 3.





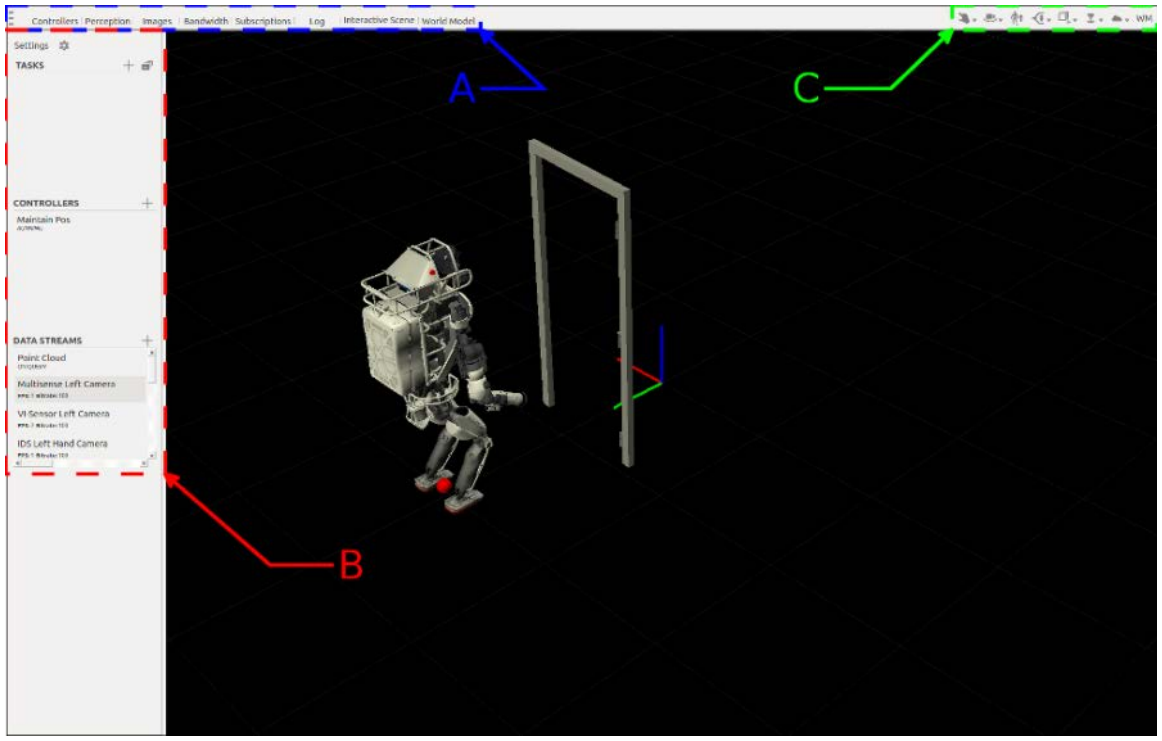


Figure 21. The UI with Annotated Elements

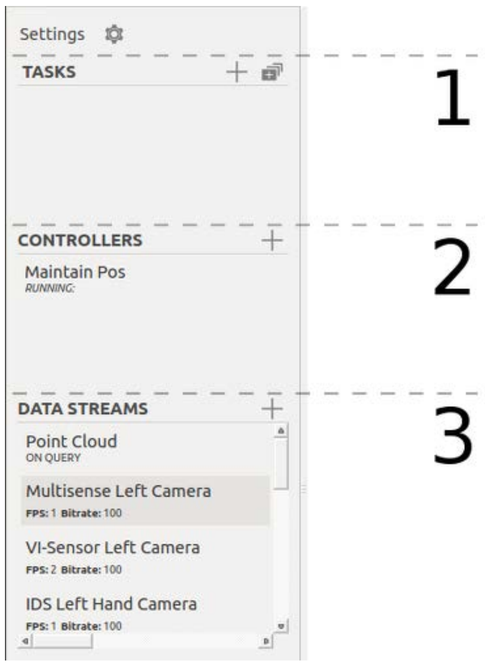


Table 3: List of Annotated UI Elements

Annotation	Description
A1	Controller Monitor Tab
A2	Adaptive Perception Manager Tab
A3	Image Streams Tab
A4	Bandwidth Monitor Tab
A5	Subscription Management Tab
A6	Logging Tab
A7	Panda Interactive Scene Tab
A8	World Model Tab
B1	Tasks Sidebar Pane
B2	Running Controllers Sidebar Pane
B3	Data Streams Sidebar Pane
C1	Panda View Interaction Quick Action
C2	Panda View Presets Quick Action
C3	Stand Up Quick Action
C4	Hand Control Quick Action
C5	Insert World Model Object Quick Action
C6	Camera Management Quick Action
C7	World Model Synchronization Quick Action

3.1.2.5.1 **Tabs.** Each UI tab focuses on a presenting a single aspect of the TROOPER system to the user. We detail the functionality of each tab.

Panda Interactive Scene. The interactive scene tab allows the user to view the robot, live sensor feeds, world model objects, and previews any planned or pending robot motion. It uses the Panda3D open source rendering engine for display. In conjunction with the various task sidebar panes mentioned later, this view allows the user to direct the motion of the robot, from placing footsteps, to indicating objects to grasp, to posing the robot by dragging targets for its end effectors.

The rendering view can be changed through mouse commands and options in the quick action bar. The robot is always rendered at its position in the world frame with its joints in the configuration currently found in the RobotModel. There is a semi-transparent 'ghost robot' which shows a desired robot location. When generating motion plans, the ghost will cycle through the plan until the operator approves or rejects the plan. The rendering features visualizations for ROS message topics such as point clouds, cost maps, OctoMaps, and convex hulls along a 2D plane. These renderers can be turned off and on through the Data Streams pane in the sidebar. Additionally, certain visual aids for the operator will be displayed when appropriate actions are being used or manually turned on: There is a semi-cylindrical "shield" that shows the limits of manipulator reachability, a variety of interactive markers for positioning objects and end effectors, footsteps that show a created walking path, and a pyramid-like line of sight indicator.

When running the robot, the operator will spend the vast majority of his or her time with this tab open. The interactive scene is shown in Figure 22.

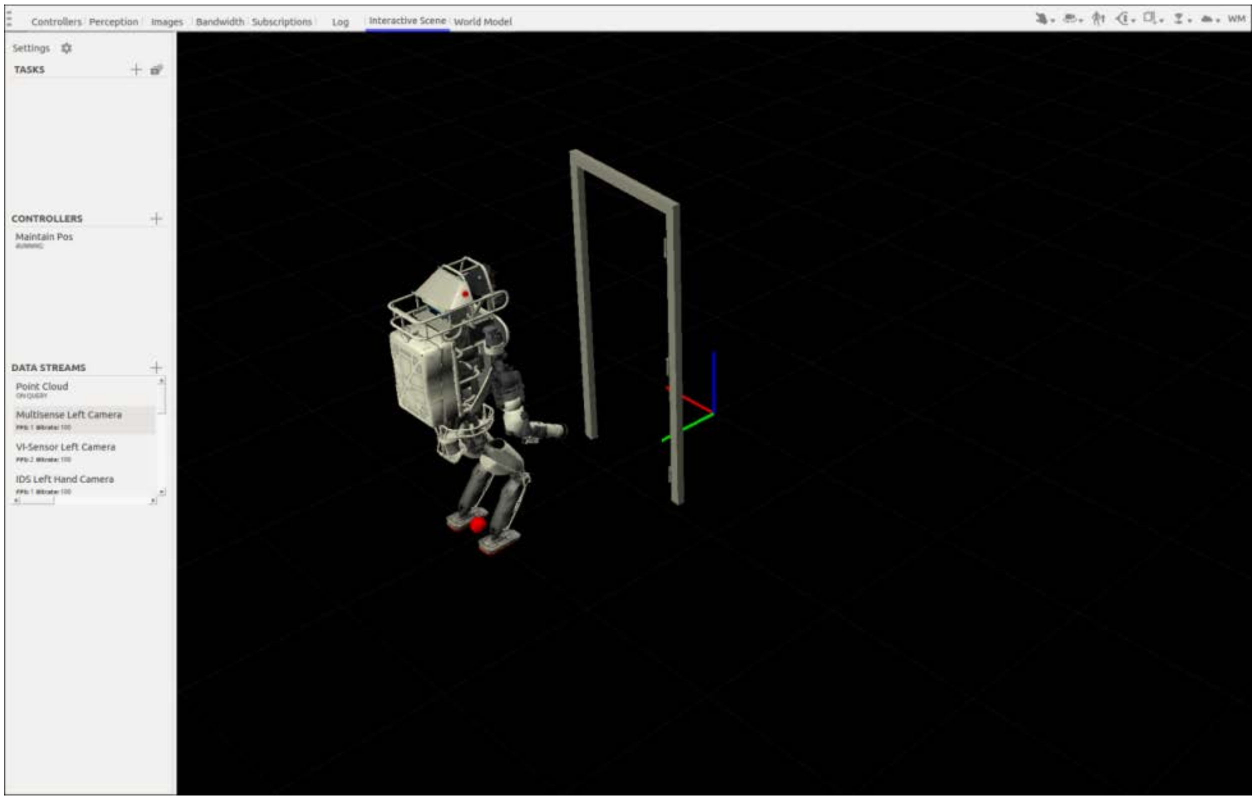
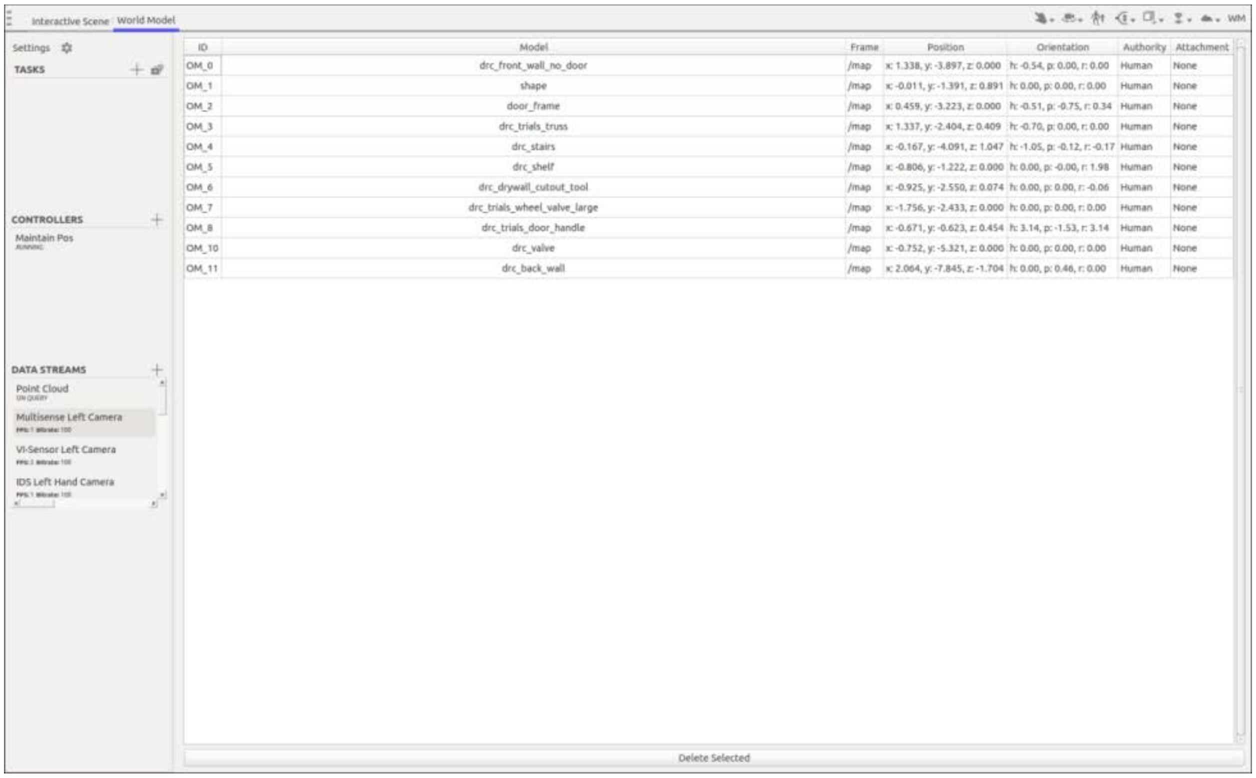


Figure 22. The Panda Interactive Scene

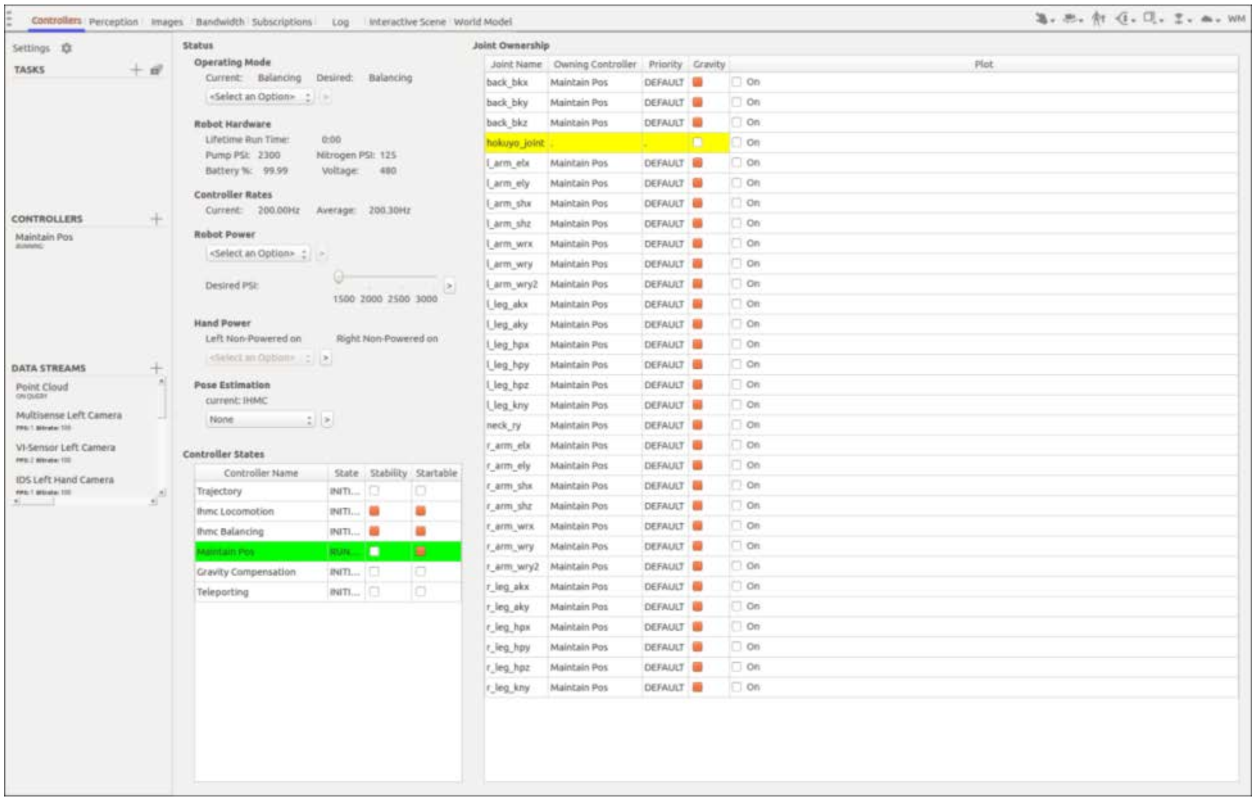
World Model. The world model tab is a simple view of the current list of world model objects and their properties. It shows the name of the object, its pose in the world reference frame, whether the human or robot has most recently updated that object in the shared world model, and which hand, if any, the object is attached to. It is mostly used for ad-hoc debugging. The world model tab is shown in Figure 23.



ID	Model	Frame	Position	Orientation	Authority	Attachment
OM_0	drc_front_wall_no_door	/map	x: 1.338, y: -3.897, z: 0.000	h: -0.54, p: 0.00, r: 0.00	Human	None
OM_1	shape	/map	x: -0.011, y: -1.391, z: 0.891	h: 0.00, p: 0.00, r: 0.00	Human	None
OM_2	door_frame	/map	x: 0.459, y: -3.223, z: 0.000	h: -0.51, p: -0.75, r: 0.34	Human	None
OM_3	drc_trials_truss	/map	x: 1.337, y: -2.404, z: 0.409	h: -0.70, p: 0.00, r: 0.00	Human	None
OM_4	drc_stairs	/map	x: -0.167, y: -4.091, z: 1.047	h: -1.05, p: -0.12, r: -0.17	Human	None
OM_5	drc_shelf	/map	x: -0.806, y: -1.222, z: 0.000	h: 0.00, p: -0.00, r: 1.98	Human	None
OM_6	drc_drywall_cutout_tool	/map	x: -0.925, y: -2.550, z: 0.074	h: 0.00, p: 0.00, r: -0.06	Human	None
OM_7	drc_trials_wheel_valve_large	/map	x: -1.756, y: -2.433, z: 0.000	h: 0.00, p: 0.00, r: 0.00	Human	None
OM_8	drc_trials_door_handle	/map	x: -0.671, y: -0.623, z: 0.454	h: 3.14, p: -1.53, r: 3.14	Human	None
OM_10	drc_valve	/map	x: -0.752, y: -5.321, z: 0.000	h: 0.00, p: 0.00, r: 0.00	Human	None
OM_11	drc_back_wall	/map	x: 2.064, y: -7.845, z: -1.704	h: 0.00, p: 0.46, r: 0.00	Human	None

Figure 23. The World Model Tab

Controller Monitor. The controller monitor tab displays statistics from the controller manager (update rate, battery capacity, and current operating mode) as well as allowing modification of many of its settings. It allows the user to enable or disable the hydraulics, change the desired pump pressure, switch pose estimation modes, and transition between different controller states. It also displays the ownership, use state, and priority of the robot's joints in the controller manager. Many of the initial commands at startup are set through this panel. The Controller Monitor is shown in Figure 24.



Maintain Pos

Point Cloud

Multisense Left Camera

VI-Sensor Left Camera

IDS Left Hand Camera

Status

Operating Mode

Current: Balancing Desired: Balancing

<Select an Option>

Robot Hardware

Lifetime Run Time: 0:00

Pump PSI: 2300 Nitrogen PSI: 125

Battery %: 99.99 Voltage: 480

Controller Rates

Current: 200.00Hz Average: 200.30Hz

Robot Power

<Select an Option>

Desired PSI: 1500 2000 2500 3000

Hand Power

Left Non-Powered on Right Non-Powered on

<Select an Option>

Pose Estimation

current: IHMC

None

Controller States

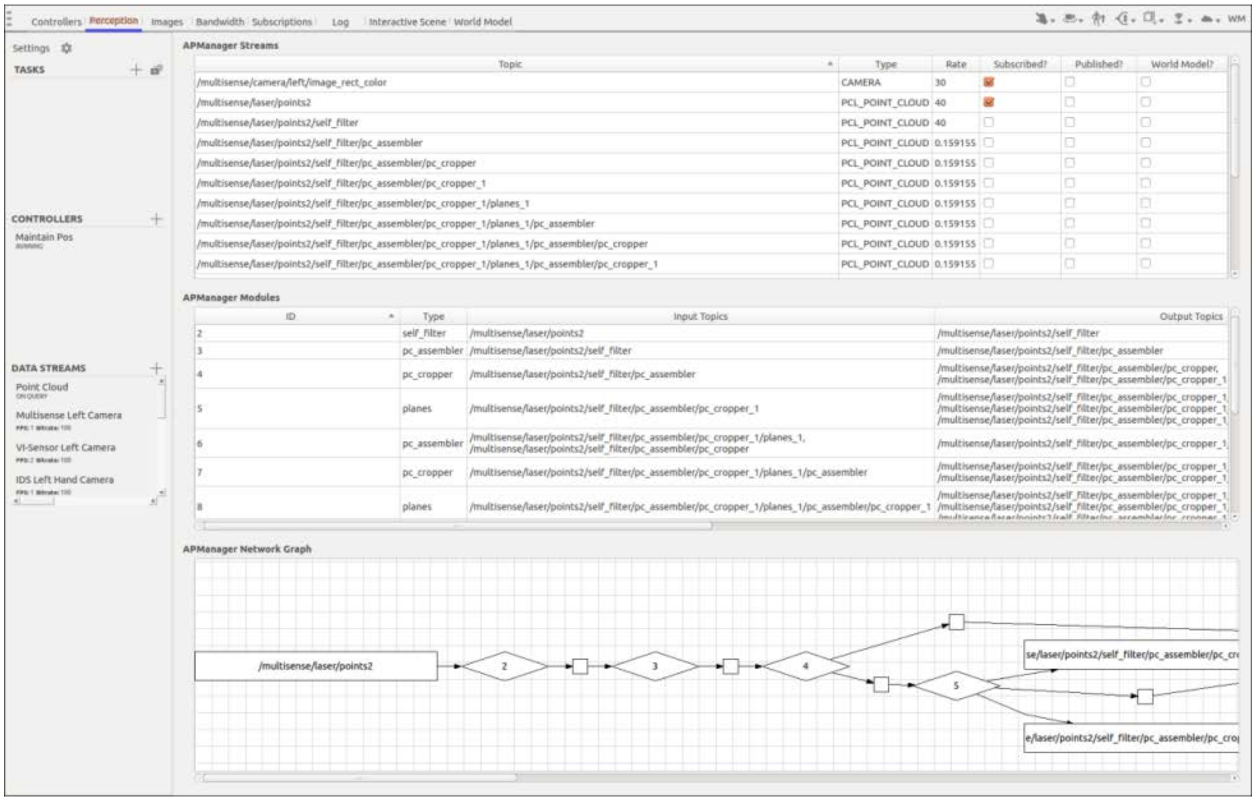
Controller Name	State	Stability	Startable
Trajectory	INITI...	☐	☐
Ihmc Locomotion	INITI...	🟧	🟧
Ihmc Balancing	INITI...	🟧	🟧
Maintain Pos	RUN...	☐	🟧
Gravity Compensation	INITI...	☐	☐
Teleporting	INITI...	☐	☐

Joint Ownership

Joint Name	Owning Controller	Priority	Gravity	Plot
back_bkx	Maintain Pos	DEFAULT	🟧	☐ On
back_bky	Maintain Pos	DEFAULT	🟧	☐ On
back_bkz	Maintain Pos	DEFAULT	🟧	☐ On
hokuyo_joint	-	-	☐	☐ On
l_arm_elx	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_ely	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_shx	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_shz	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_wrx	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_wry	Maintain Pos	DEFAULT	🟧	☐ On
l_arm_wry2	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_akx	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_aky	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_hpx	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_hpy	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_hpz	Maintain Pos	DEFAULT	🟧	☐ On
l_leg_kny	Maintain Pos	DEFAULT	🟧	☐ On
neck_ry	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_elx	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_ely	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_shx	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_shz	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_wrx	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_wry	Maintain Pos	DEFAULT	🟧	☐ On
r_arm_wry2	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_akx	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_aky	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_hpx	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_hpy	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_hpz	Maintain Pos	DEFAULT	🟧	☐ On
r_leg_kny	Maintain Pos	DEFAULT	🟧	☐ On

Figure 24. The Controller Monitor Tab

Adaptive Perception Manager. The adaptive perception manager tab shows statistics published from the adaptive perception manager. It provides three categories of information: a table of sensor streams (topic, type, rate, and publishing flags); a table of active perception processing modules (id, type, input topics, and output topics); and a network graph showing how the streams and modules connect. Visualization of certain sensor streams in the interactive scene tab can be turned on and off via right clicking the corresponding component on the network graph. The adaptive perception manager tab is shown in Figure 25.



APManager Streams

Topic	Type	Rate	Subscribed?	Published?	World Model?
/multisense/camera/left/image_rect_color	CAMERA	30	☑	☐	☐
/multisense/laser/points2	PCL_POINT_CLOUD	40	☑	☐	☐
/multisense/laser/points2/self_filter	PCL_POINT_CLOUD	40	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1/pc_assembler	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1/pc_assembler/pc_cropper	PCL_POINT_CLOUD	0.159155	☐	☐	☐
/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1/pc_assembler/pc_cropper_1	PCL_POINT_CLOUD	0.159155	☐	☐	☐

APManager Modules

ID	Type	Input Topics	Output Topics
2	self_filter	/multisense/laser/points2	/multisense/laser/points2/self_filter
3	pc_assembler	/multisense/laser/points2/self_filter	/multisense/laser/points2/self_filter/pc_assembler
4	pc_cropper	/multisense/laser/points2/self_filter/pc_assembler	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper, /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1
5	planes	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1 /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1 /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1
6	pc_assembler	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1, /multisense/laser/points2/self_filter/pc_assembler/pc_cropper	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1
7	pc_cropper	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1/pc_assembler	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1 /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1
8	planes	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1/planes_1/pc_assembler/pc_cropper_1	/multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1 /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1 /multisense/laser/points2/self_filter/pc_assembler/pc_cropper_1

Figure 25. The Adaptive Perception Manager Tab

Image Streams. The image streams tab allows the operator to view the various image streams that the robot's sensors capture. A simple dropdown box allows the user to select which stream should be rendered. This tab is used during driving as the primary source of information. It is also used during manipulation and locomotion as a supplement to the interactive scene – typically the left screen will display the image stream and the right will have the interactive scene. This allows the operator to teleoperate or diagnose reasons for an automated step to fail. The image streams tab is shown in Figure 26.

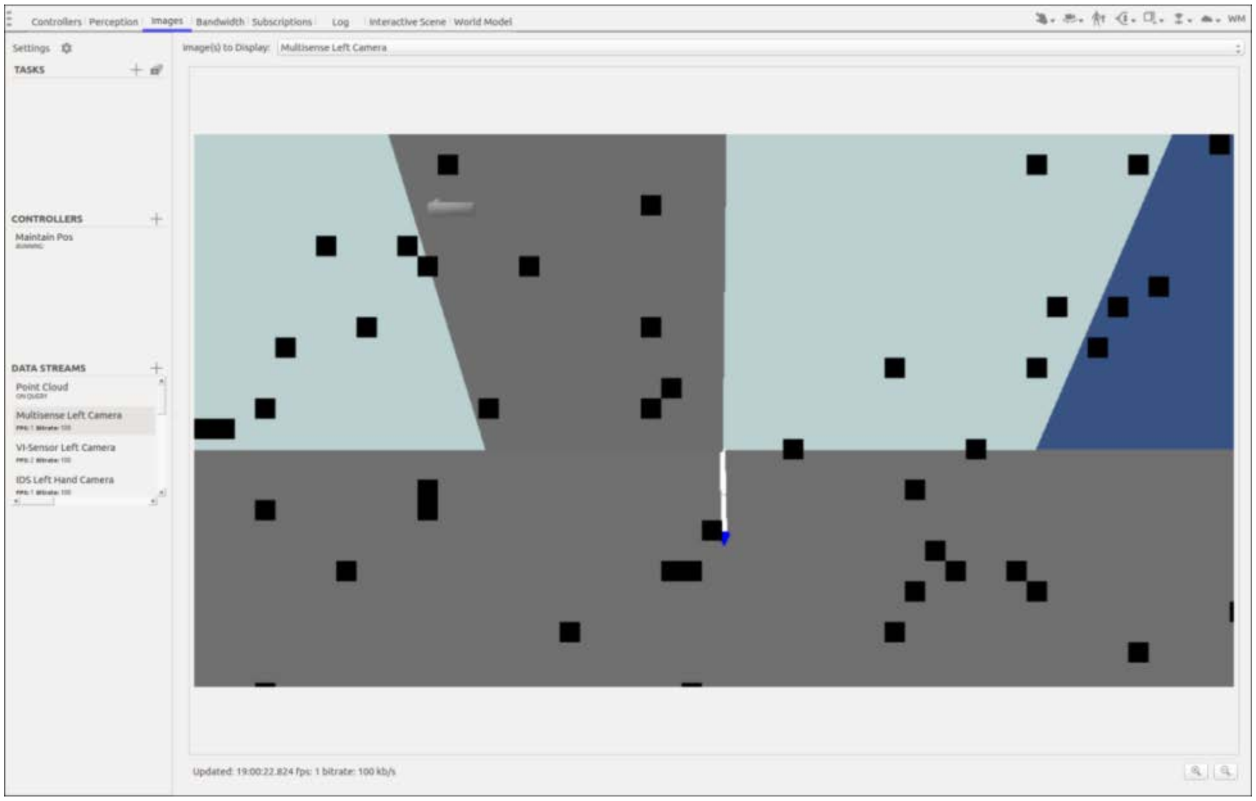
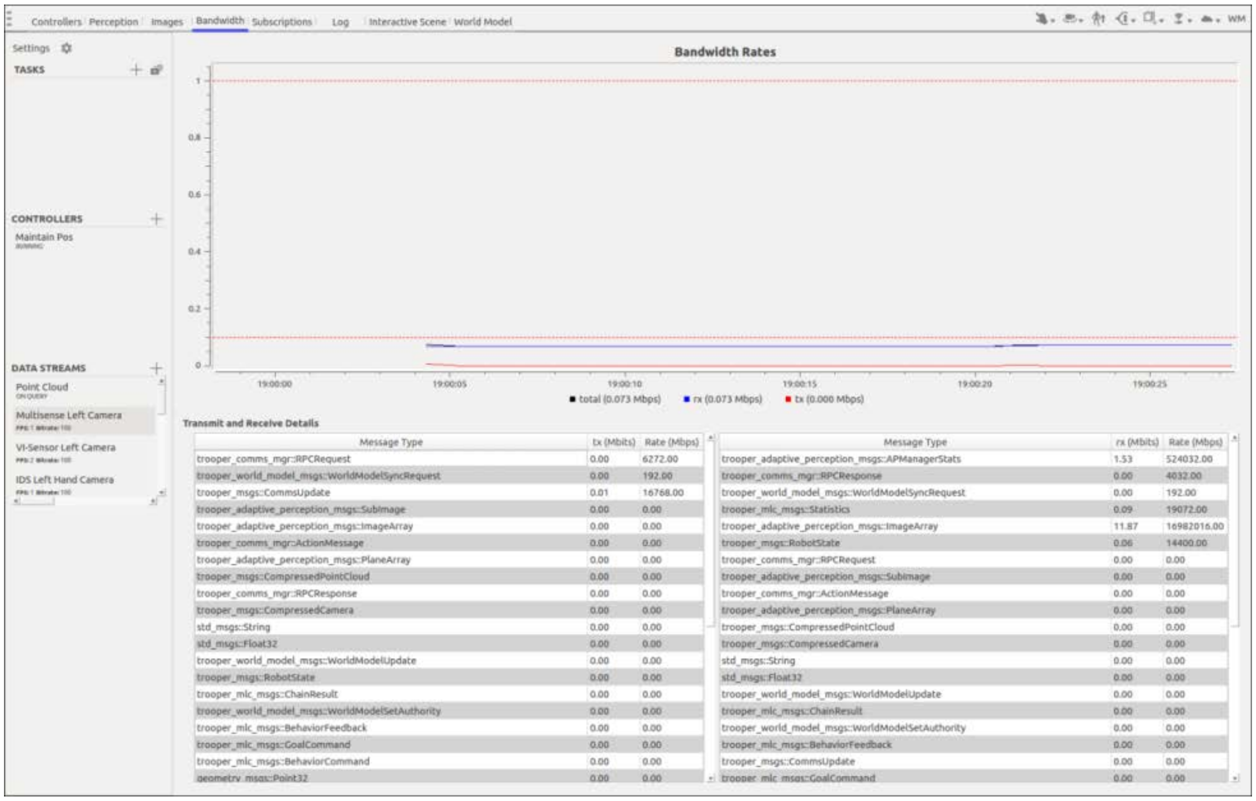


Figure 26. The Image Streams Tab

Bandwidth Monitor. The bandwidth monitor tab shows statistics from the communications manager. It graphs the bandwidth transmitted and received as a function of time. It also has views of messages types being transmitted and received along with the transmission rate thereof. Given the strict communications restrictions, this tab was mainly used to identify bandwidth intensive processes in our communications structure. The bandwidth monitor tab is shown in Figure 27.



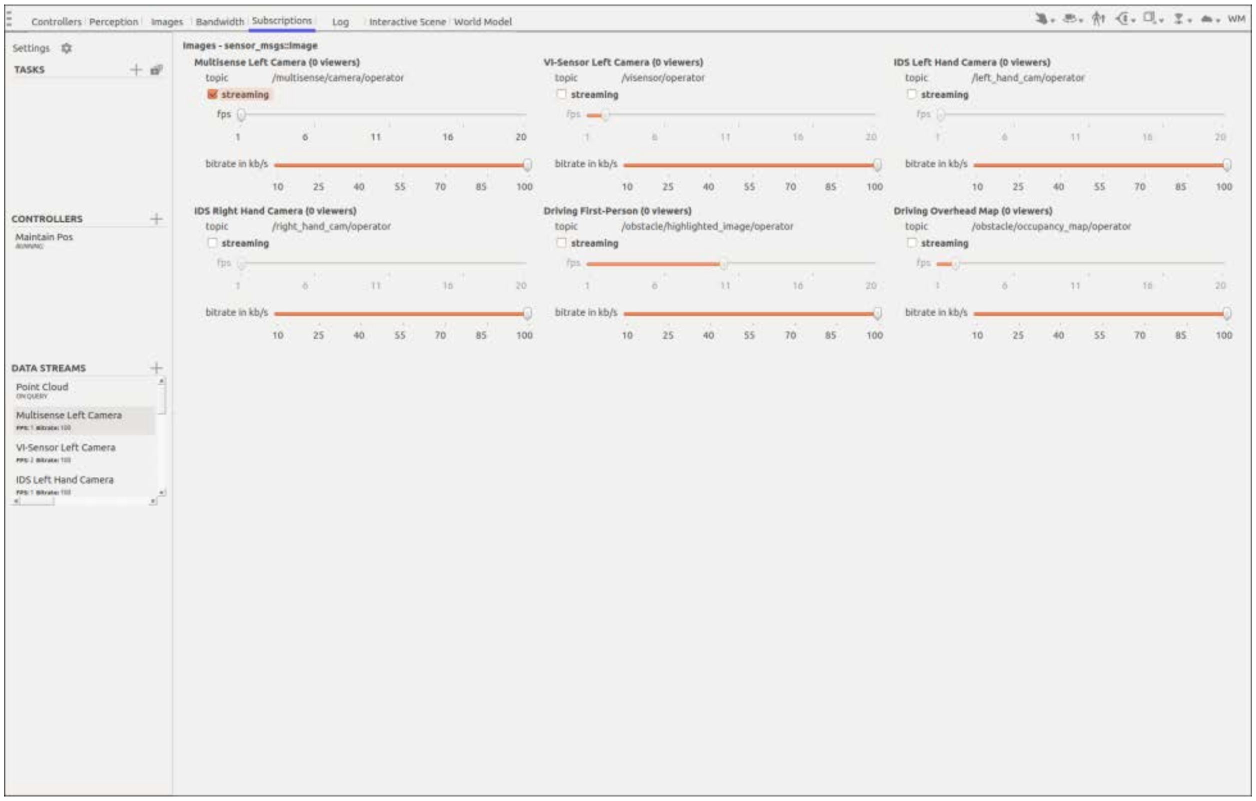
Transmit and Receive Details

Message Type	tx (Mbits)	Rate (Mbps)
trooper_comms_mgr::RPCRequest	0.00	6272.00
trooper_world_model_msgs::WorldModelSyncRequest	0.00	192.00
trooper_msgs::CommsUpdate	0.01	16768.00
trooper_adaptive_perception_msgs::SubImage	0.00	0.00
trooper_adaptive_perception_msgs::ImageArray	0.00	0.00
trooper_comms_mgr::ActionMessage	0.00	0.00
trooper_adaptive_perception_msgs::PlaneArray	0.00	0.00
trooper_msgs::CompressedPointCloud	0.00	0.00
trooper_comms_mgr::RPCResponse	0.00	0.00
trooper_msgs::CompressedCamera	0.00	0.00
std_msgs::String	0.00	0.00
std_msgs::Float32	0.00	0.00
trooper_world_model_msgs::WorldModelUpdate	0.00	0.00
trooper_msgs::RobotState	0.00	0.00
trooper_mlc_msgs::ChainResult	0.00	0.00
trooper_world_model_msgs::WorldModelSetAuthority	0.00	0.00
trooper_mlc_msgs::BehaviorFeedback	0.00	0.00
trooper_mlc_msgs::GoalCommand	0.00	0.00
trooper_mlc_msgs::BehaviorCommand	0.00	0.00
geometry_msgs::Point32	0.00	0.00

Message Type	rx (Mbits)	Rate (Mbps)
trooper_adaptive_perception_msgs::APManagerStats	1.53	524032.00
trooper_comms_mgr::RPCResponse	0.00	4032.00
trooper_world_model_msgs::WorldModelSyncRequest	0.00	192.00
trooper_mlc_msgs::Statistics	0.09	19072.00
trooper_adaptive_perception_msgs::ImageArray	11.87	16982016.00
trooper_msgs::RobotState	0.06	14400.00
trooper_comms_mgr::RPCRequest	0.00	0.00
trooper_adaptive_perception_msgs::SubImage	0.00	0.00
trooper_comms_mgr::ActionMessage	0.00	0.00
trooper_adaptive_perception_msgs::PlaneArray	0.00	0.00
trooper_msgs::CompressedPointCloud	0.00	0.00
trooper_msgs::CompressedCamera	0.00	0.00
std_msgs::String	0.00	0.00
std_msgs::Float32	0.00	0.00
trooper_world_model_msgs::WorldModelUpdate	0.00	0.00
trooper_mlc_msgs::ChainResult	0.00	0.00
trooper_world_model_msgs::WorldModelSetAuthority	0.00	0.00
trooper_mlc_msgs::BehaviorFeedback	0.00	0.00
trooper_msgs::CommsUpdate	0.00	0.00
trooper_mlc_msgs::GoalCommand	0.00	0.00

Figure 27. The Bandwidth Monitor Tab

Subscriptions. The subscriptions tab, shown in Figure 28, displays the current status of sensor streams and allows enabling/disabling streams. It also allows updating the FPS and bitrate.



Maintain Pos
RUNNING

Point Cloud
ON QUERY

Multisense Left Camera
FPS: 1 Bitrate: 100

VI-Sensor Left Camera
FPS: 2 Bitrate: 100

IDS Left Hand Camera
FPS: 1 Bitrate: 100

Images - sensor_msgs::Image

Multisense Left Camera (0 viewers)
topic /multisense/camera/operator
[✓ streaming]
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

VI-Sensor Left Camera (0 viewers)
topic /visensor/operator
streaming
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

IDS Left Hand Camera (0 viewers)
topic /left_hand_cam/operator
streaming
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

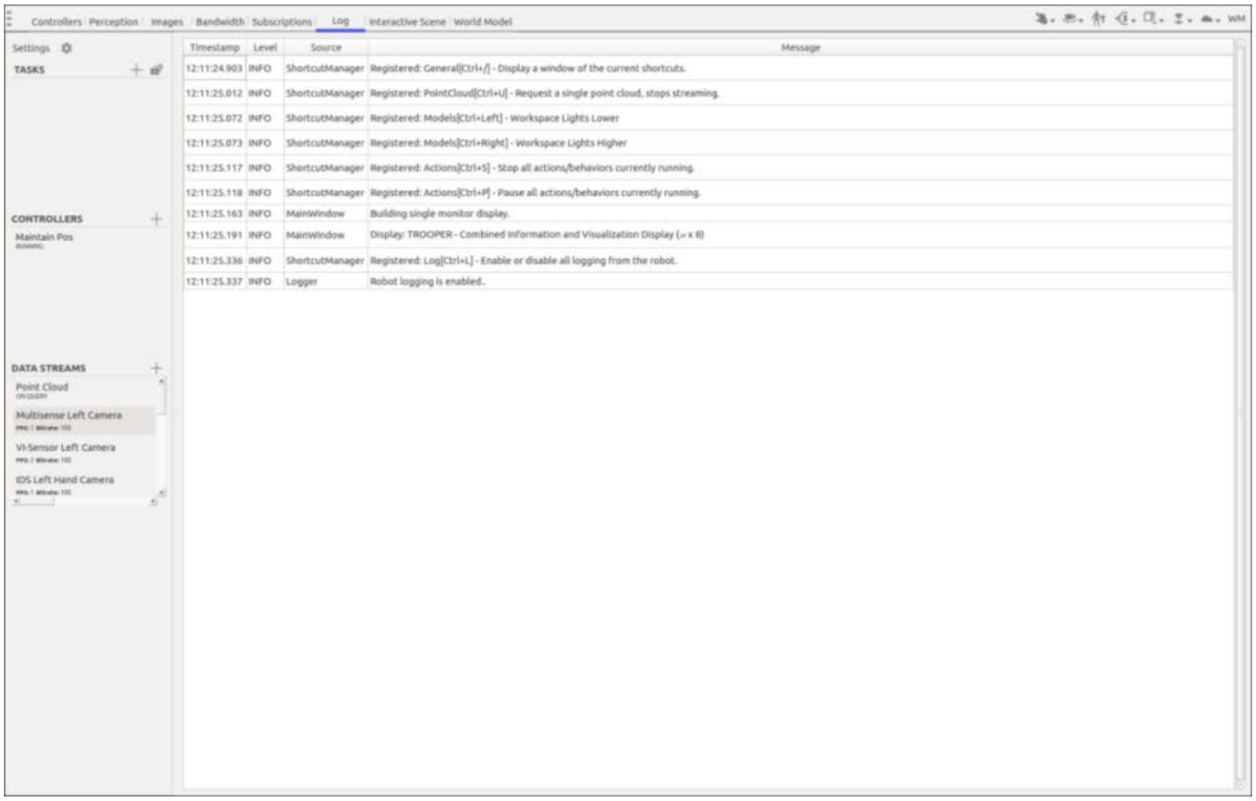
IDS Right Hand Camera (0 viewers)
topic /right_hand_cam/operator
streaming
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

Driving First-Person (0 viewers)
topic /obstacle/highlighted_image/operator
streaming
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

Driving Overhead Map (0 viewers)
topic /obstacle/occupancy_map/operator
streaming
fps 1 6 11 16 20
bitrate in kb/s 10 25 40 55 70 85 100

Figure 28. The Subscriptions Tab

Logging. The logging tab, shown in Figure 29, displays a table of logged messages from the ROS system on the OCU. These messages are color coded on Level. Important messages also show up as pop-ups in the upper right of the interactive scene tab.



Timestamp	Level	Source	Message
12:11:24.903	INFO	ShortcutManager	Registered: General[Ctrl+/] - Display a window of the current shortcuts.
12:11:25.012	INFO	ShortcutManager	Registered: PointCloud[Ctrl+U] - Request a single point cloud, stops streaming.
12:11:25.072	INFO	ShortcutManager	Registered: Models[Ctrl+Left] - Workspace Lights Lower
12:11:25.073	INFO	ShortcutManager	Registered: Models[Ctrl+Right] - Workspace Lights Higher
12:11:25.117	INFO	ShortcutManager	Registered: Actions[Ctrl+S] - Stop all actions/behaviors currently running.
12:11:25.118	INFO	ShortcutManager	Registered: Actions[Ctrl+P] - Pause all actions/behaviors currently running.
12:11:25.163	INFO	MainWindow	Building single monitor display.
12:11:25.191	INFO	MainWindow	Display: TROOPER - Combined Information and Visualization Display (# x 8)
12:11:25.336	INFO	ShortcutManager	Registered: Log[Ctrl+L] - Enable or disable all logging from the robot.
12:11:25.337	INFO	Logger	Robot logging is enabled..

Figure 29. The Logging Tab

3.1.2.5.3 **Sidebar Panes.** The UI sidebar panes provide the operator access to common functionality required regardless of the active tab. They allow for adding and updating tasks, enabling or disabling controllers, and modifying sensor subscriptions and perception processing. The panes panel is shown in Figure 30.

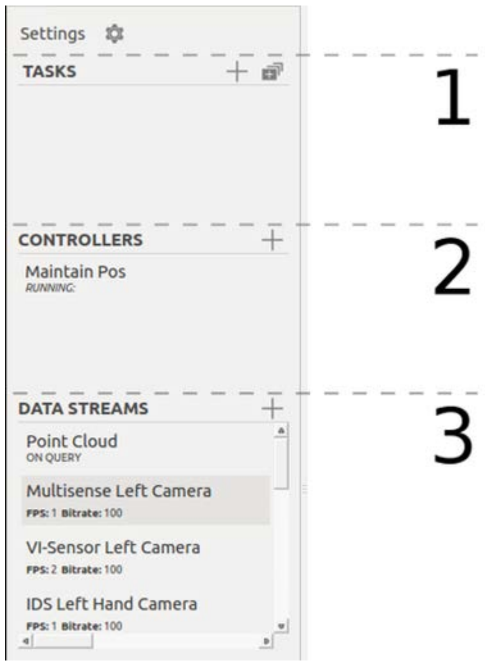


Figure 30. Sidebar Panes

Tasks. The tasks pane (1) lists all current reasoner-level tasks in the system. Single tasks may be added by clicking the plus sign button on the top right. (Tasks may also be added by right clicking on the robot in the interactive scene tab.) Task chains may be added by clicking the box stack to the right of the plus sign. Either option brings up the task link widget to the left of the interactive scene. If a chain was selected, first the user specifies the goals for the chain and clicks the approve button. Then a task ribbon is created above the interactive scene and monitors in use are displayed in a bay at the bottom. The task ribbon has a chevron for each task in the chain. If a task is running, the gear icon to the left of the task name will change to a right arrow. If a chevron is clicked, the task parameters will be displayed in the task link widget. Monitors can be set to a desired condition manually by right clicking.

Controllers. The controllers pane (2) lists all running controllers. By right clicking on these controllers, they can be started, stopped, or paused as on the controller monitor tab. The plus button in the upper right allows the user to start controllers that are not currently running.

Data Streams. The data streams pane (3) displays three different types of rows:

- The “Point Cloud” row can be right clicked to switch the set of streams that are rendered in the interactive scene.
- The image stream rows are actively captured image streams. These streams can be stopped by right clicking on the row and selecting stop. They can be added through the plus button in the upper right.
- The adaptive perception stream rows are the same streams that are listed in the adaptive perception tab. The rendering of these streams can be turned off and on via a right click menu.

3.1.2.5.4 Quick Action Bar. The quick action bar is located on the upper right of the UI window and contains menus and buttons for common tasks. The quick action bar is shown in Figure 31.

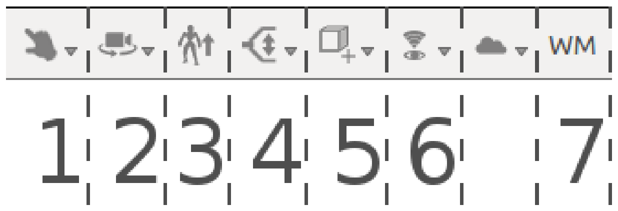


Figure 31. Quick Action Bar

1. **Panda View Interaction.** The view interaction dropdown (1) allows the user to change the left click function of the panda interactive scene.
2. **Panda View Presets.** The view presets dropdown (2) allows the user to switch to a selection of viewpoints in the panda interactive scene.
3. **Stand up.** The stand button (3) toggles between walking and balancing modes. If the robot has been squatting in balancing mode, the transition will cause it to return to a walking height.
4. **Hand Control.** The hand control dropdown (4) allows the user to open and close the left hand.
5. **World Model Objects.** The world model objects menu (5) allows the user to insert various world model objects. Once one is selected, the user can click anywhere in the scene to add the object there.
6. **Camera Management.** The camera management menu (6) allows the user to show various aids in the panda interactive scene and creates popup windows for the different camera streams.
7. **World Model Sync.** The world model sync button (7) attempts to synchronize the world model on the OCU with the world model on the RCU.

3.1.2.6 Simulation. Our choice of which simulator to use for each DRC task split cleanly along the lines of supported features of each simulation package. Simulating upper body motions and anything requiring hands was done using DRCSim and Gazebo. Simulating walking was done using IHMC's Simulation Construction Set (SCS). We had mockups of the DRC Finals setup in both simulators, as well as the ability to spawn the robot in front of each task setup.

Gazebo Simulation. We had used Gazebo DRCSim simulation extensively for the VRC and DRC Trials in 2013. For the VRC, the simulation had been tuned to allow it to run at near-real-time rates, albeit with simplified robot collision models and masses tuned for stability rather than resemblance to the physical Atlas. For the DRC Trials, we received updated robot models and a gait model from Boston Dynamics for the walking behavior they provided, though these dropped the simulation real-time factor to under 30 percent of real-time when running with the rest of the TROOPER system on one machine. The DRC Finals 2015 Atlas is able to walk in Gazebo simulation, though it is much more unsteady than the physical Atlas and is unable to handle any steps or otherwise uneven terrain.

The Gazebo simulation supports collision checking all simulated objects, though often with separate visual and collision models (either different fidelity meshes or just simple shapes for the collision model). It uses Open Dynamics Engine (ODE) as underlying physics simulation, though it now has nominal support for other physics engines. That said, DRCSim robot behaviors have been tuned for ODE and do not function using the other engines.

The Gazebo simulation supported all available hand types and hand physics (namely collision checking), which meant that we used it to test manipulation planning. Unfortunately, it was never very robust for interacting with simulated objects after the VRC; the tendency was for the simulation to go unstable when solving contact constraints for multi-fingered hands grasping objects. The simulation also did not model the soft contacts inherent in real robotic grippers, which we believe to be a critical factor in stable grasping. Gazebo also simulated all sensors present on Atlas, complete with noise, in large part because support has been developed in ROS. Thus, we conducted our perception manager tests in Gazebo when the real robot was unavailable. The LIDAR simulation used a ray-casting model with additional Gaussian noise, and we found this to be consistent with hardware experiments. The visual complexity of the simulated environments was insufficient, and thus the stereo simulation was not used for testing. Future robot simulation should leverage physically based rendering techniques and environmental assets developed by the video game and motion picture industries. Lastly, Gazebo supported writing plugins, which we used to allow robot teleportation to speed up simulation tests. Atlas in gazebo simulation is shown in Figure 32.

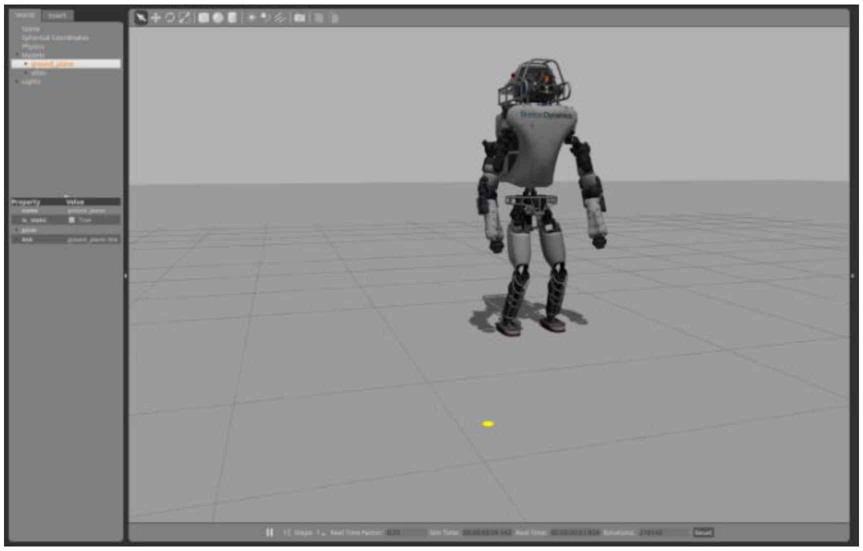


Figure 32. Atlas Gazebo Simulation

IHMC Simulation Construction Set (SCS). IHMC generously released their controllers and simulation package to any Atlas teams who chose to use them. Gazebo did not support the IHMC controllers; the only way to test them in simulation was to use IHMC's bundled Simulation Construction Set (SCS), shown in Figure 33. IHMC tuned their SCS simulation to match the performance of the controller on the real robot, and as such their simulated Atlas was capable of walking over cinderblocks and other uneven terrain. That said, IHMC's primary focus was on walking and led to the exclusion of other key features. The SCS did not support simulating hands and thus was unusable for practicing manipulation tasks. It also only performed collision checking between the feet and the ground; so if it had had hands, they would not have been able to grasp anything. Lastly, it did not simulate LIDAR noise and the camera model simulation did not accurately match the MultiSense SL, making it unable to test much of our perception system.

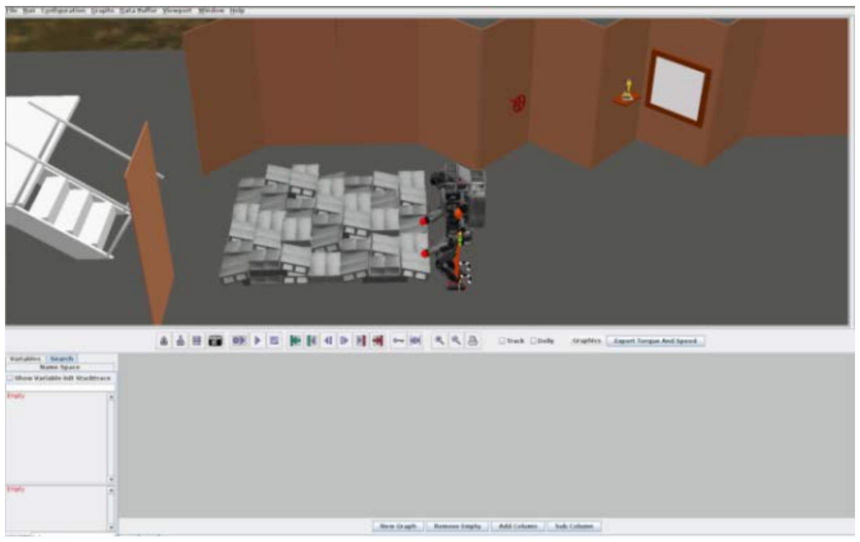


Figure 33. Atlas in IHMC Simulation Construction Set

3.2 Capabilities

This section details the TROOPER system's control, planning, and perception functionality. Control ranges in sophistication from controlling the position or torque at each joint to sophisticated balancing and walking. Planning includes both motion planning for the upper body, enabling grasping and manipulation, as well as walking trajectory planning. The perception category is broad, including our perception manager framework and components therein, such as localization, mapping, plane detection, and object detection.

3.2.1 Controls. When Atlas first arrived, we were given controllers produced by Boston Dynamics. These controllers had specific modes, including walking, manipulation, and user. Walking mode did not allow control of the upper body; manipulation mode kept the robot balanced while moving the upper body and changing the pelvis height; and user mode provided the operator with joint-level control over the entire robot. For the DRC Trials, we chose to include our own balancing controller, initially written for the Virtual Robotics Challenge. This controller operated in the user mode provided by the Boston Dynamics software. Lastly, through an agreement with DARPA, IHMC released their controllers that completely replace those provided by Boston Dynamics. We modified our system to be compatible with both control frameworks and ended up using the IHMC controllers in competition. *We were the only team to do so for the DRC Finals.*

3.2.1.1 TROOPER Balancing Controller. This is the controller we implemented and used in the VRC for balancing and walking, as well as in the DRC Trials for balancing. For a robot to statically balance, the projection of the center-of-mass to a plane perpendicular to the gravity vector (for simplicity, we will call this the ground plane) must lie within the support polygon of the robot. For a robot on flat ground, the support polygon can be defined as the convex hull of all contacts with the ground. When frictional contacts are made upon sloped surfaces, the center of mass must lie above a nonlinear convex set that depends on the properties of the contacts. In this work we do not use zero moment point (ZMP) methods, but rather a balancing approach adapted from the field of robot grasping, as demonstrated by Christian Ott at the German Aerospace Center, DLR.

We begin by outlining the balancing controller described in (Ott, Roa, & Hirzinger, 2011). The method is based on frictional grasping; forces **f** are applied at contact points **P** to generate a net wrench **F** on the on the object being grasped sufficient to keep it restrained. In the case of balancing, the desired wrench is applied to the robot center of mass (COM) and is used to track a desired pelvis orientation and COM location; i.e., the robot should remain relatively upright, compensating for gravity, with its projected COM within the support polygon. The contact forces used to do this are those on the feet of the robot.

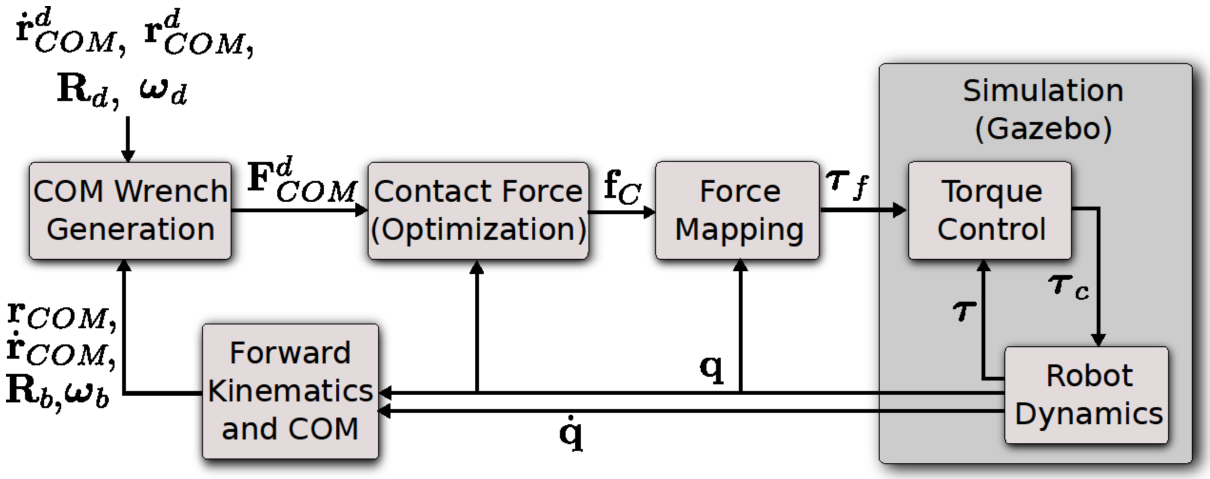


Figure 34. Overview of the Balancing Controller

Center of Mass Position and Posture Controller. The desired center of mass (COM) force is given by:

$$\mathrm{f}_c^d = \boldsymbol{mg} - \boldsymbol{K_p}(\boldsymbol{r} - \boldsymbol{r^d}) - \boldsymbol{K_D}(\dot{\boldsymbol{r}} - \dot{\boldsymbol{r}}^{\boldsymbol{d}}) \quad (2)$$

where the gravity compensation term contains *m* the total mass of the robot and **g** the gravity vector, while the latter terms are a PD feedback law to drive the center of mass to a desired location. $\boldsymbol{K}_P, \boldsymbol{K}_D > 0$ are proportional and differential gain matrices, and $\boldsymbol{r}^d, \dot{\boldsymbol{r}}^d$ are the desired position and velocity of the center of mass.

The desired COM torque is used to track a desired pelvis orientation. Let $\boldsymbol{R}_b$be the current and $\boldsymbol{R}_d$ be the desired pelvis orientation. From the quaternion representation of $\boldsymbol{R}_d^T\boldsymbol{R}_b = (x, y, z, w)$, let $\delta = w$ and $\epsilon = (x, y, z)$. Then an orientation controller for pelvis orientation is given by:

$$\boldsymbol{\tau^d} = -\boldsymbol{R_b}(\boldsymbol{2}(\boldsymbol{\delta I} + \hat{\boldsymbol{\epsilon}})\boldsymbol{K_r\epsilon} + \boldsymbol{D_r}(\boldsymbol{\omega} - \boldsymbol{\omega^d})) \tag{3}$$

where $\boldsymbol{K}_r, \boldsymbol{D}_r$ are symmetric, positive definite stiffness and damping matrices, respectively. This controller acts as a damped spring to align the current orientation $\boldsymbol{R}_b$ with the desired $\boldsymbol{R}_d$. Together, $\mathbf{f}^d$ and $\boldsymbol{\tau}^d$ comprise the desired wrench acting at the center of mass, $\boldsymbol{F}_d$

Contact Force Distribution. Now that we have a desired wrench to apply at the COM, we need to find the contact forces at the feet that will produce it. The following is a brief review of multi-contact grasping.

The contact forces at the feet are subject to the positivity restriction; they can push but not pull the ground. Coulomb's friction model is used, stating that the contacts do not slip when:

$$\boldsymbol{f^t} \leq \boldsymbol{\mu f^n} \tag{4}$$

where f^n is the magnitude of the normal component of the contact force, f^t the tangential component, and μ the coefficient of friction. In $\mathbb{R}^3$ this restricts the set of allowable contact forces to a cone called the friction cone, whose axis is along the surface normal with a semi-angle of $\emptyset = \tan^{-1}\mu$.

The total wrench on the object, $\boldsymbol{F}_O$ is the sum of the wrenches from all of the contacts expressed in the object's coordinate frame, O. For a system with η contacts, let $\mathbf{f}_c$ be a vector stacking all the individual contact forces, $\mathbf{f}_c = (\mathbf{f}_1 \ldots \mathbf{f}_\eta)^T$. Then the expression for the total wrench is:

$$\boldsymbol{F_O} = \boldsymbol{G}\mathrm{f}_{\boldsymbol{c}} \tag{5}$$

where **G** is the grasp map, mapping the wrenches from the local contact point coordinate frame P_i to the object frame O and multiplying by the wrench basis characterizing the contact model. With all frictional point contacts, the grasp map becomes:

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{R_{p_1}} & \cdots & \boldsymbol{R_{p_\eta}} \\ \vdots & \ddots & \vdots \\ \hat{\boldsymbol{r}}_{\boldsymbol{p_1}}\boldsymbol{R_{p_1}} & \cdots & \hat{\boldsymbol{r}}_{\boldsymbol{p_\eta}}\boldsymbol{R_{p_\eta}} \end{pmatrix} \tag{6}$$

where $\boldsymbol{R}_{p_i}$ and $\hat{\boldsymbol{r}}_{p_i}$ represent the orientation and cross product matrix of the position of the contact *i* in the object reference frame *O*.

When standing, the grasp map **G** is known and we need to solve for the contact forces $\mathbf{f}_c$ at the feet. Because the problem is underconstrained, we cast this problem as a quadratic optimization:

$$\underset{q\in\mathbb{R}^n}{\mathbf{arg\,min}}\, \boldsymbol{\alpha_1}\left\|\boldsymbol{F^d} - \boldsymbol{G}\mathrm{f}_{\boldsymbol{c}}\right\|_2^2 + \boldsymbol{\alpha_2}\,\mathrm{f}_{\boldsymbol{c}}^{\boldsymbol{T}}\mathrm{f}_{\boldsymbol{c}} \tag{7}$$

subject to:

$$\mathrm{f}_{c_i} = \sum_{j=1}^{k} \boldsymbol{\sigma}_{ij}\boldsymbol{n}_{ij}, \quad \boldsymbol{\sigma}_{ij} \geq \mathbf{0}, \qquad \boldsymbol{i} = \mathbf{1}, \dots, \boldsymbol{\eta} \tag{8}$$

The constraints above come from approximating the friction cone as a polyhedron; $\boldsymbol{n}_{ij}$ is the j-th edge of the convex cone at the i-th contact point. The first term of the cost function penalizes distance between the effective center of mass wrench $\boldsymbol{F} = \boldsymbol{G}\boldsymbol{f}_c$ and the desired center of mass wrench; the second term attempts to evenly distribute the contact forces. Weights α_1 and α_2 are chosen such that $\alpha_1 \gg \alpha_2 > 0$.

Now that we have the contact forces at the feet, we can find the equivalent wrenches in each foot's frame. The wrenches can then be mapped to joint torques using the Jacobian for each leg, using:

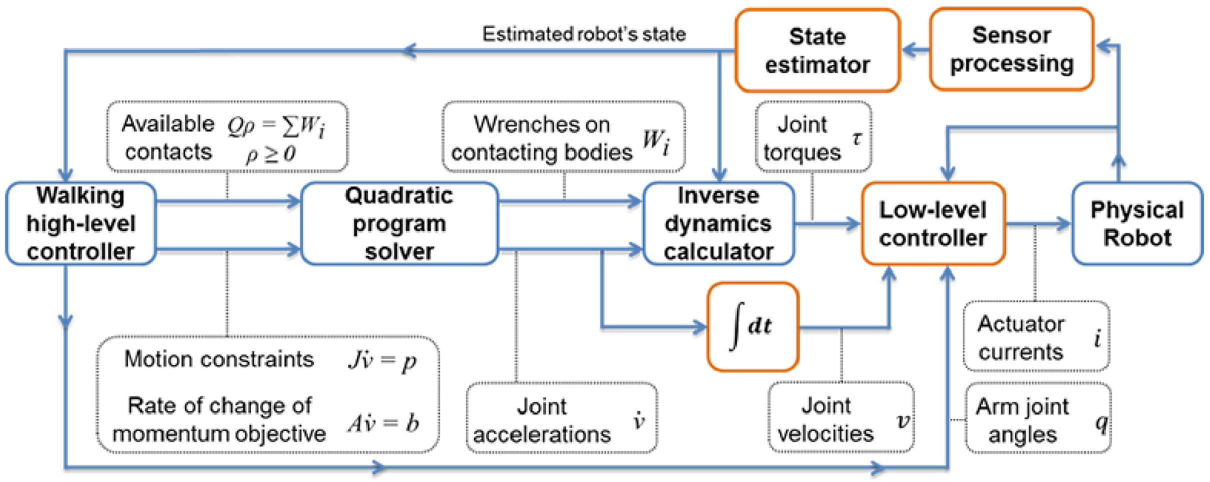
$$\boldsymbol{J}_c = \boldsymbol{J}^b - \boldsymbol{R}_{FP}\boldsymbol{J} \tag{9}$$

$$\boldsymbol{\tau} = \boldsymbol{J}_c^T \boldsymbol{F} \tag{10}$$

where $\boldsymbol{J}^b$ is the body Jacobian for each foot with the pelvis as the root link, $\boldsymbol{R}_{FP}$ is the rotation from foot to pelvis, $\boldsymbol{J}$ the center of mass Jacobian for each leg, and $\boldsymbol{\tau}$ the joint torque vector.

3.2.1.2 IHMC Whole Body Controller. The IHMC controller ran on two separate computers inside Atlas. One machine was dedicated to running the high speed control loop and ran a real-time Linux kernel. The other machine handled ROS messaging (both sending and receiving messages from the rest of the system).

Figure 35 below shows the components running in the real-time control loop. The walking high-level controller interprets the commands from ROS messages. The resulting desired motions and actions are sent to the quadratic program (QP) solver in the form of objectives. The QP solver translates them into desired joint accelerations and contact wrenches. The inverse dynamics calculator takes in the accelerations and wrenches and calculates the desired robot joint torques. The low-level controller controls the individual joints and attempts to track a desired torque trajectory.



Estimated robot's state
State estimator
Sensor processing
Available contacts $Q\rho = \sum W_i$ $\rho \geq 0$
Wrenches on contacting bodies W_i
Joint torques τ
Walking high-level controller
Quadratic program solver
Inverse dynamics calculator
Low-level controller
Physical Robot
$\int dt$
Actuator currents i
Motion constraints $J\dot{v} = p$
Rate of change of momentum objective $A\dot{v} = b$
Joint accelerations $\dot{v}$
Joint velocities v
Arm joint angles q

Figure 35. The IHMC Whole Body Controller

There are two scenarios which bypass the flow shown in the diagram. In the case of the arms, the QP solver is not used and the desired joint angle and velocity trajectories are sent to the low level controller (which will track these instead of joint torques in this case). Additionally, when the robot is in whole body position control mode (rather than 'walking' mode, which is actively balancing), all the joint angles are tracked directly by the low level controller.

More details on the workings of the controller can be found in (Pratt, 2015) At the high level, the robot can be placed in ‘walking mode' which is actively balancing, or 'whole body position mode' which allows all joint angles to be specified by the user. Whole body position mode was only used while in the vehicle. The walking mode had multiple interfaces to control different portions of the robot:

- Chest orientation – specified chest orientation in world frame (we ran forward kinematics using desired joint angles to fill this out)
- Head orientation – specified head orientation in world frame (we ran forward kinematics using desired joint angles to fill this out)
- Pelvis height – control the pelvis height relative to a reference value
- Arm control – specify either a joint angle trajectory (positions and velocities) for an arm or specify a desired end effector pose and let the controller handle the approach
- Footstep placement – specify a path of footsteps to take. Allows specifying world frame foot placement and walking gait parameters
- Desired pump PSI – allowed using lower pump pressures (<2000psi) while only walking, while ramping up to higher pressures (>2300psi) when using the upper body to perform manipulation tasks.

3.2.2 Planning. We consider planning to include: how to pose the robot body to achieve goals such as end effector placement; how to generate goal poses that are valid for grasping or otherwise interacting with objects; and how to create a motion trajectory that will get us from our current configuration to that goal configuration. We utilize single-chain and whole-body inverse kinematics solvers, task space regions, and bi-directional rapidly exploring random trees to provide these functionalities.

3.2.2.1 Inverse Kinematics. Solving the inverse kinematics problem for a single end effector on Atlas, calculating the joint angles required to place the end effector at a desired pose with respect to the world frame, was an 11-dimensional problem. Specifically, it required solving for the 3 back joints angles (roll, pitch, and yaw of the upper torso with respect to the pelvis), 7 arm joint angles, and what we referred to as a ‘pseudo-joint’, a prismatic joint representing the height of the pelvis. Solving whole-body inverse kinematics involved solving for the full 30 degrees-of-freedom; using a nonlinear optimization solver allowed us to solve for multiple end effector positions and apply other constraints.

Initial (Traditional) IK Approach. In general, the fastest method to solve inverse kinematics is to use a closed-form solution, which can be done as long as the manipulator has six or fewer degrees of freedom. This also immediately gives feedback as to the existence of a solution. With redundant degrees of freedom (greater than six), we lose the ability to use closed-form solutions directly. In cases where there are few additional degrees of freedom, one can perform a line search over ranges of the additional joints, solving the closed-form equations at each point until a solution is found. This can be quite slow, however, depending on the discretization of the additional freedoms. Another option, and one that works for any number of additional freedoms, is to compute a local IK solution using Jacobian-based iterative solvers. Though they can be used to solve for tree topologies, at their heart these algorithms solve for a single chain at time (and enforce constraints between iterations).

For the initial, 6 degree-of-freedom arm version of Atlas, our IK approach was to combine an analytic IK solver with an iterative, Jacobian-based IK solver. We used IKFAST, part of the Open Robotics Automation Virtual Environment (Diankov, 2010) to generate closed-form solution for the 6 degree-of-freedom arm. For queries involving the back or squatting, we used Kinematics and Dynamics Library's (KDL's) iterative, Jacobian-based IK solver (Smits, 2015). To speed up the iterative search, we created a library of IK solutions with respect to the upper torso for use in seeding the iterative solver. The IK solutions were stored by 6-dimensional pose coordinates. We stored approximately 50,000 solutions in a lookup table, spaced in a 3D grid every 10cm and every 90 degrees. These IK solutions were encoded as a k-d tree for efficient lookup. For solving higher-dimensional queries, those including the back joints and pelvis heights, we iterated over pelvis heights and back joint yaws – each time looking for a close match IK solution in the k-d tree. If one existed, we would use that to seed the IK solver. To simplify the problem, we assumed that the pelvis roll and pitch remained zero.

Atlas ran its own balancing controller onboard (initially one that we implemented and later one provided by IHMC as part of their whole body control package). Thus, when we would move the upper body, the lower part of the body would shift to keep the robot's center of mass in a fixed location. If the motion involved leaning or large arm displacements, the mass distribution could shift dramatically, leading to a pelvis displacement of over 10cm. This motion needed to be accounted for to accurately solve the inverse kinematics problem and get the end effector to a desired location. The solution was to add another layer of iteration, solving the IK solution for a fixed pelvis position, calculating the updated center of mass, then shifting the pelvis position to keep the center of mass position (in the X-Y plane) unchanged – the center of mass height was allowed to change as the pelvis height was controlled separately.

The downsides of this traditional IK method are that it inherently acts upon a single chain and does not handle arbitrary constraints well. Tree topologies can be handled by solving one chain at a time and then adjusting the constraints on child chains in the next iteration. Handling the balancing constraints likewise required large modifications. We also used weighted damped least squares to penalize using certain joints, such as the back joints, but had no other mechanism to punish deviation from a desired reference configuration. That said, with our system we were able to produce single end effector, 11 degree-of-freedom IK solutions at > 60Hz. Representative solutions are shown in Figure 36. Note that these only move the left arm; the joint angles of the right arm remain unchanged.

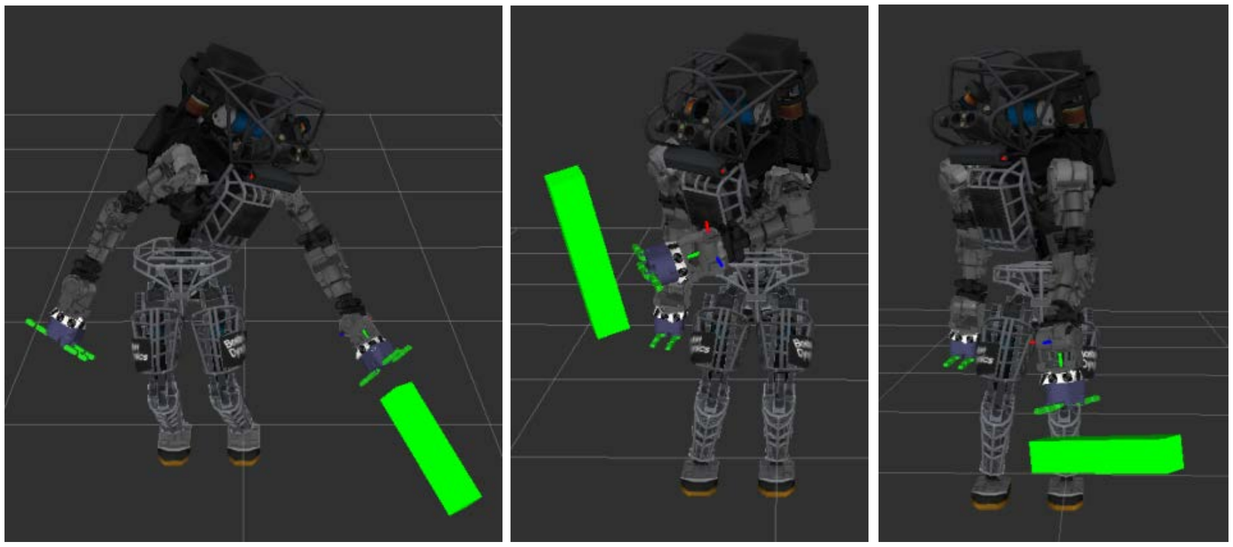


Figure 36. Three Inverse Kinematic Solutions for Box Pre-Grasps

Whole Body (Drake) IK. The primary restriction of our initial inverse kinematics pipeline was its restriction to solving a single chain and its inability to easily handle arbitrary constraints. It could handle solving for one end effector position, keeping the center of mass in the same location, but would not be of use if the end effector pose was only partially satisfied or if we wanted to constrain both end effectors. To overcome these limitations and allow for solving whole-body postures, we adopted the inverse kinematics framework from MIT is Drake package **(Tedrake, 2014)**. While Drake is primarily written in MATLAB, there is a C++ version of the robot model and inverse kinematics solvers.

The Drake solver casts inverse kinematics as a nonlinear optimization problem and uses sequential quadratic programming (SQP) to solve for a local minimum. Our integrated version is capable of producing whole-body configurations for Atlas at > 20Hz; that is solving for Atlas's 30 degrees-of-freedom (technically 36, given that the pelvis is treated as a floating base, free to move in 6 dimensions). The optimizer's objective function was to minimize the weighted distance from a nominal configuration:

$$\underset{q\in\mathbb{R}^n}{\arg\min}(q_{nom} - q)^T W(q_{nom} - q)$$

subject to

$$f_i(q) \leq b_i, \qquad i = 1, \ldots, m$$

where W is the weighting matrix, q_{nom} the nominal robot configuration in generalized coordinates (so for Atlas, $n = 36$), q the solution robot configuration, and f_i a set of kinematic constraint functions.

The user is free to choose from as many or as few of the allowed supported constraint types; the solver takes in an arbitrary length vector of these constraints. The allowed constraint types are:

- Joint limits
- End effector position (with allowed upper and lower bounds)
- End effector orientation (with allowed tolerance)
- Cone constraints ('gaze' constraint allowing rotation about an axis)
- Distance between bodies
- Center of mass position

- Symmetry

These constraints can be combined easily in any query without requiring any modification to the algorithm code, a huge advantage compared to the traditional approach. Typically, we specify different constraints on each hand while fixing the location of the feet and fixing the X-Y location of the center of mass while allowing it to move vertically. The floating pelvis is effectively constrained to have zero roll and pitch by placing a large weight on orientation deviations. In the case of turning a valve, we used gaze constraints to constrain the stick to remain perpendicular to the valve face while allowing rotation about the stick axis.

Interactive IK. We created the ability to pose the robot on the fly and give the operator real-time feedback on IK solutions prior to motion execution. This allowed for fine, detailed adjustment of robot poses. Additionally, we allowed clicking on an object to quickly place the hand a certain distance removed along the object's outward normal. Figure 37 shows the interactive IK in action. The left image shows the 6 degree-of-freedom markers on the wrists that allow the user to pose the robot and have the solver give solutions in real time. On the right, shift-clicking the pole places the end effector in a grasp pose near the contact normal specified by the mouse click.

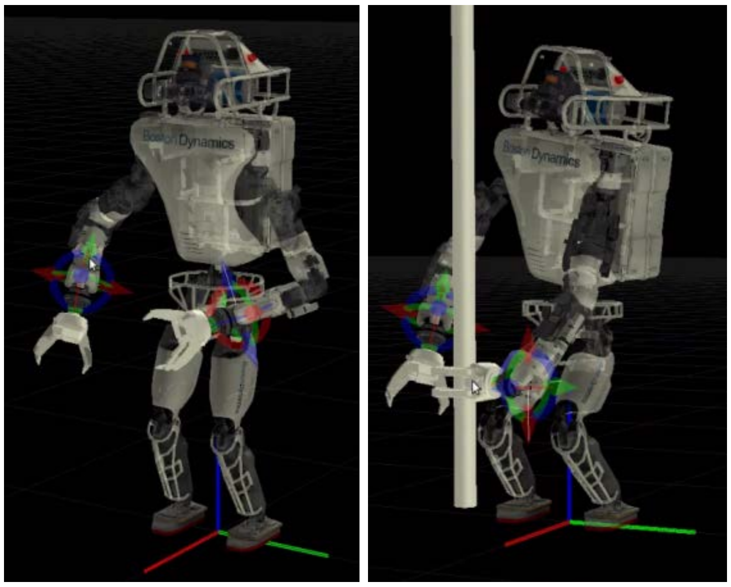


Figure 37. Interactive Inverse Kinematics

3.2.2.3 Manipulation Planning. To keep the TROOPER system general-purpose, we eschewed hard-coded or object-specific grasps. Instead, grasps were generated dynamically at run time using the currently encountered object models. This allows flexibility and enables the system to grasp any object at any time, regardless of whether or not the object has been previously encountered.

Task Space Regions. Our overall planning framework utilized Task Space Regions (TSRs) to encode graspable regions around objects (Berenson, Srinivasa, & Kuffner, 2011). TSRs describe end effector constraint sets as subsets of special Euclidean group SE(3). This representation combines the constraints upon the end effector with the available affordances; a sample from the TSR is guaranteed to be a valid grasp (unless in collision).

TSRs consist of 3 parts: (Note that T_b^a is the homogeneous transformation matrix from a to *b*.)

- T_{tsr}^{obj}– transform from the object origin to the TSR origin
- T_{ee}^{tsr}– transform for the end effector offset in the TSR origin frame
- B^{tsr}– a 6 x 2 matrix of the bounds in the coordinates of the TSR origin frame

$$\boldsymbol{B^{w}} = \begin{bmatrix} \boldsymbol{x}_{\mathbf{min}} & \boldsymbol{x}_{\mathbf{max}} \\ \boldsymbol{y}_{\mathbf{min}} & \boldsymbol{y}_{\mathbf{max}} \\ \boldsymbol{z}_{\mathbf{min}} & \boldsymbol{z}_{\mathbf{max}} \\ \boldsymbol{\sigma}_{\mathbf{min}} & \boldsymbol{\sigma}_{\mathbf{max}} \\ \boldsymbol{\theta}_{\mathbf{min}} & \boldsymbol{\theta}_{\mathbf{max}} \\ \boldsymbol{\varphi}_{\mathbf{min}} & \boldsymbol{\varphi}_{\mathbf{max}} \end{bmatrix} \tag{11}$$

Matrix B^{w} bounds to rotation and translation about the TSR origin frame, which is related to the object origin by T_{tsr}^{obj}. The end effector offset, T_{ee}^{tsr}, easily allows for the use of different hands.

TSRs have some very useful properties. It is easy to calculate the distance from a TSR, which is useful when monitoring whether or not an object is graspable before attempting to close the hand around it. They are also straightforward to sample, requiring only generating 6 random numbers from a specified range (or 7, in the case of selecting one TSR to sample from in larger set of TSRs). As the constraint manifold is often lower-dimensional than the state space, the ability to sample from the constraint manifold makes sampling-based planners practical.

Object Decomposition. Complex objects are simplified and decomposed into simple primitive shapes: cylinders, spheres, and cuboids. Each of these simple shapes has an associated set of Task Space Regions that encode the types of allowed grasps. For example, consider a hand grasping a cylinder. The hand may grab the side of the cylinder with the thumb at the top of the hand or with the hand rotated 180 degrees so the thumb is at the bottom. The hand may also grasp the top or bottom of the cylinder (the hand would be allowed to rotate freely about the palm in this case). Thus, we get 4 TSRs representing the set of valid grasps on a cylinder. Figure 38 shows valid grasps obtained by regularly sampling tasks space regions for simple shapes. Each red dot represents the palm location (with origin as shown on the right half of the figure) of a grasp.

Additionally, we compare the dimensions of the object against the maximum grasp aperture of the hand. The following graphic shows possible grasp locations on a set of simple shapes arrived at by regularly sampling from the TSRs. Each red dot represents the palm frame origin of a grasp.

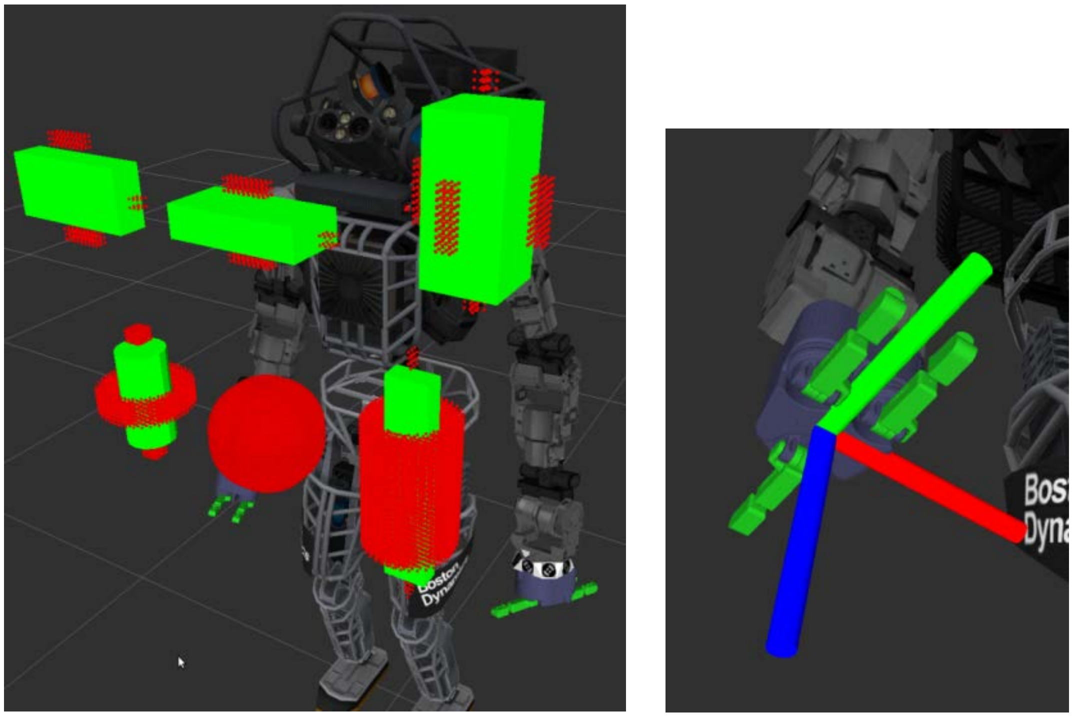


Figure 38. Task Space Regions

3.2.2.4 Motion Planning. We sample from the Task Space Regions comprising the object to be manipulated. Each TSR sample is equivalent to a full 6 degree-of-freedom end effector pose. For each sample, we generated not only a final grasp pose (near the object) but also a pregrasp pose 10cm further out from the object. We first check that an inverse kinematics solution exists for the pregrasp, then perform a collision check, and last verify that a similar inverse kinematics solution exists for the final grasp. The final grasp is not collision checked because the hand will always be in collision with the object. Our collision checking routines make use of the Flexible Collision Library (Pan, Chitta, & Manocha, 2012) to perform both broad and narrow phase checks of simple geometric primitives such as boxes and cylinders.

Once we have a valid goal, we generate a motion plan from the robot's current configuration to the pregrasp pose. We use the RRTConnect algorithm as implemented in the Open Motion Planning Library (Sucan, Moll, & Kavraki, 2012). Afterwards, the resulting trajectory is smoothed using shortcut smoothing and splined. The balancing controller is simulated during planning; for each new sampled configuration, the center of mass location is updated and the robot pelvis shifted to keep the center of mass in the same world-frame position. This is done prior to collision checking the new configuration. Once we have a valid plan to the pregrasp, we interpolate to reach the grasp pose. The RRT plan is combined with the interpolated portion.

Had we required additional constraints during motion planning, such as keeping a held cup of water upright so as not to spill, these would have been encoded in the TSR for the object and used to generate samples for the RRT planner. This would have been effectively the Constrained Bi-Directional RRT algorithm (Berenson, Srinivasa, & Kuffner, 2011).

3.2.2.5 Footstep Planning. We developed both simple footstep interpolation and an A* based planner for generating walking paths. Unlike the A* planner, the interpolation does not perform collision checking and, hence, the plans it generates must be validated by the operator.

Footstep Interpolator. We developed simple footstep interpolation schemes that generate sequences of foot placement goals that will move the robot from point A to point B without any consideration for avoiding obstacles in the environment. We use a constant stride length to determine the spacing between each step. Our initial interpolation scheme linearly interpolates both X-Y position in the ground plane and rotation about the Z-axis, then samples steps using our stride length. This interpolation method results in the robot tripping and falling for most goal positions. A more conservative scheme involves first turning in place to face the goal position, walking forward toward the goal, then turning in place again to achieve the goal orientation. This method was very reliable, but required a substantial number of steps. Both methods are shown in Figure 39; the left image shows the linear interpolation and the right image shows straight-stepping to the same goal position then turning to achieve the desired orientation.

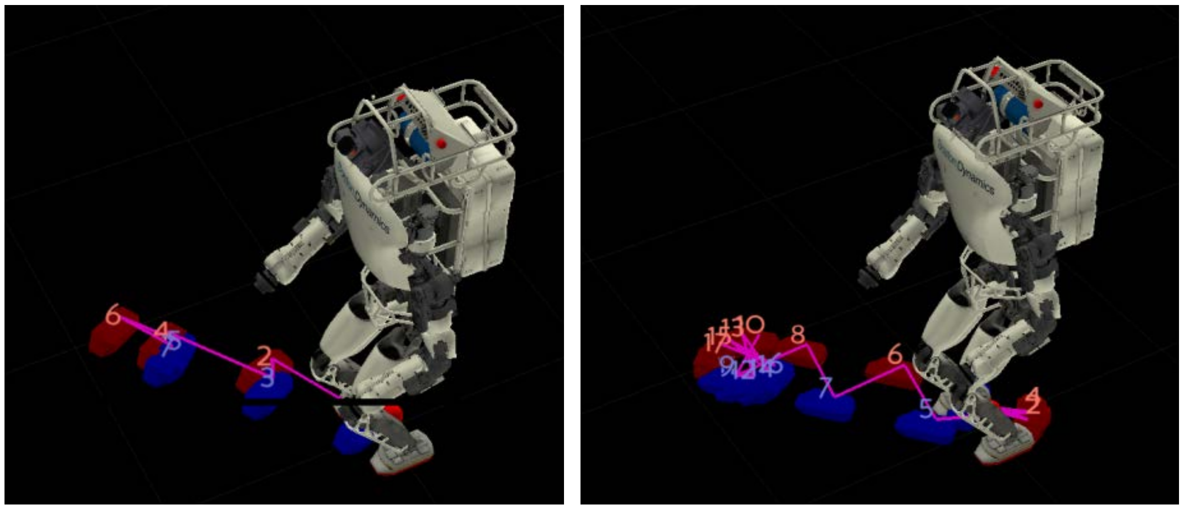


Figure 39. Interpolated Footsteps

The cause of failure in the original interpolation scheme was that the Atlas robot is kinematically incapable of taking a step in a “toe-in” configuration where the two feet orientations create an intersection point in front of the robot, illustrated in Figure 40 with a safe foot configuration on the left and an unsafe configuration on the right.

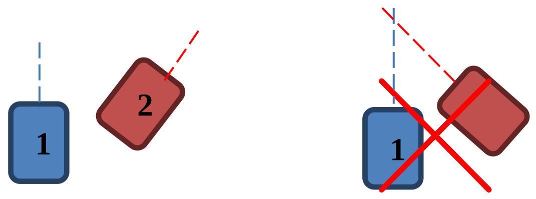


Figure 40. Safe and Unsafe Foot Configurations

Our final method avoids toe-in configurations with few footsteps by leading with the proper foot. Unlike our conservative turn-in-place method, this approach initially turns in place only when the heading toward the goal position deviates more than 40 degrees from the current heading. Otherwise, it steps off with the right foot to head to the right and vice versa. The same process is used in reverse when reaching the goal position. This scheme results in footstep patterns that resemble natural human behavior.

We not only support patterns that walk forward to the goal, but also backwards and side-stepping. We generate paths with each of these 3 methods then reorder them to prioritize the path requiring the fewest steps. We often found that a backwards walk takes fewer steps than turning around and walking forward, but we rarely found the sidestepping scheme to be beneficial other than for very short distances. This is because the stride length for side-stepping is more constrained than in the forward direction. Forward and backward stepping plans are shown in Figure 41.

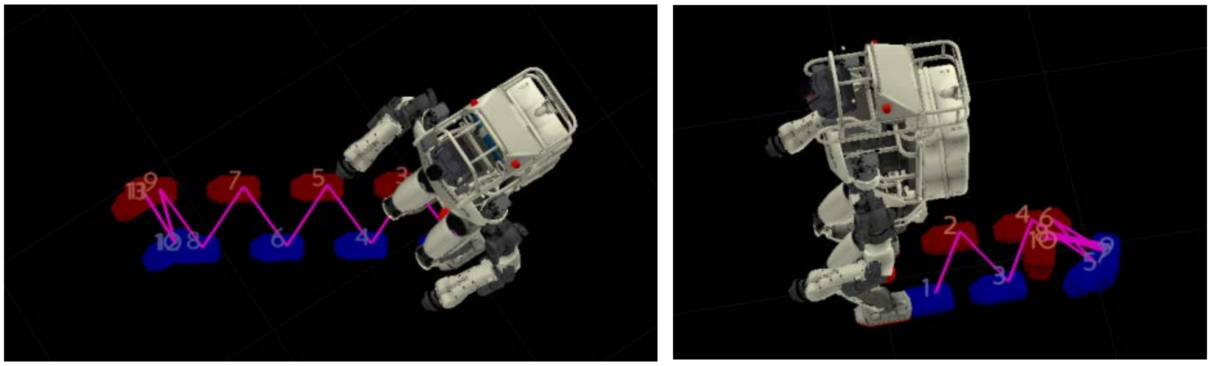


Figure 41. Forward and Backward Interpolation

Future research will use terrain maps to adapt these interpolated steps to conform to the environment. This will avoid a slow global search problem in favor of fast local adaptation of an efficient seeded set of steps.

A* Footstep Planner. For intelligent footstep planning through terrain and obstacles, we use a discrete search approach with the A* algorithm. This took in 3-DoF start/end poses and a map of anticipated costs for each square of terrain at a specified resolution. Each node allowed transitions to a new node selected by the final X-Y plane location of the step foot. Two nodes were considered equivalent if they had the same X-Y position for both the step foot and the support foot. The support foot is the step foot of the node from which the current node transitioned. Filtering and cost depend on the initial and final positions of the step foot and the position and yaw of the support foot. (Step foot yaw is aligned with the line between initial and final step foot positions, rotated by 180 degrees if greater than absolute 90 degrees from the support foot yaw, and finally set to the support foot yaw if "toed in".) In terms of complexity for A*, basic transitions happen on a 2d space, equality happens on a 4d space, and analysis happens on a 7d space. This separation allows us to build and maintain our open and closed lists quickly without substantial quality loss.

The path cost function included weighted measures of:

- Total number of steps taken
- Traversability cost at the new step position
- Squared error from desired stance width (length of normal from support foot axis to the new foot position)
- Squared error from the length of the last step

These measures ideally combine to create a path that is quick, smooth, and on safe ground. However, they do nothing to assure kinematic safety in terms of ZMP. The naïve transition function allows the new foot to be evaluated to be anywhere within a maximum step distance of the support foot. In order to create a path that was safe to traverse, we had to filter our transition function. Five metrics proved important bases upon which to filter:

1. The shortest distance between the support foot center and the line between the initial and final positions of the step foot.
2. The X and Y intercepts of the boundary line of safe absolute difference between the initial center of mass and the support foot.
3. The X and Y intercepts of the boundary line of safe absolute difference between the support foot and the final center of mass.

The first metric is used to prevent collisions between the support leg/foot and the step leg/foot. The minimum distance between the center of the support foot and any point on the line between the initial and final step foot positions must be greater than the maximum diameter of the foot. The other four metrics are used to qualify how able the support leg kinematic chain will be able to control the transition between one-legged ZMP and two-legged ZMP.
In each transition, the support leg experiences angular momentum similar to:

$$\vec{\boldsymbol{p}}(\boldsymbol{t}) = \vec{\boldsymbol{p}}(\boldsymbol{0}) + \int \vec{\boldsymbol{R}}_{\boldsymbol{CoM}}(\boldsymbol{t}) \times \left[\vec{\boldsymbol{g}} + \vec{\boldsymbol{F}}_{\boldsymbol{leg}}(\boldsymbol{t})\right] \boldsymbol{dt} \tag{12}$$

And thus the leg must be able to provide force such that:

$$\vec{\boldsymbol{R}}_{\boldsymbol{CoM}}(\boldsymbol{t}) \times \vec{\boldsymbol{F}}_{\boldsymbol{leg}}(\boldsymbol{t}) - \vec{\boldsymbol{R}}_{\boldsymbol{CoM}}(\boldsymbol{t}) \times \vec{\boldsymbol{g}} = \vec{\boldsymbol{M}}(\boldsymbol{t}) \tag{13}$$

Which provides the set of equations:

$$\boldsymbol{R}_{\boldsymbol{CoM,y}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,z}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{z}}\right] - \boldsymbol{R}_{\boldsymbol{CoM,z}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,y}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{y}}\right] = \boldsymbol{M}_{\boldsymbol{x}}(\boldsymbol{t}) \tag{14}$$

$$\boldsymbol{R}_{\boldsymbol{CoM,z}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,x}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{x}}\right] - \boldsymbol{R}_{\boldsymbol{CoM,x}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,z}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{z}}\right] = \boldsymbol{M}_{\boldsymbol{y}}(\boldsymbol{t}) \tag{15}$$

$$\boldsymbol{R}_{\boldsymbol{CoM,x}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,y}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{y}}\right] - \boldsymbol{R}_{\boldsymbol{CoM,y}}(\boldsymbol{t}) \cdot \left[\boldsymbol{F}_{\boldsymbol{leg,x}}(\boldsymbol{t}) + \boldsymbol{g}_{\boldsymbol{x}}\right] = \boldsymbol{M}_{\boldsymbol{z}}(\boldsymbol{t}) \tag{16}$$

These equations can be simplified by several assumptions. Since we assume that our frame is decently aligned with the normal of the earth, we say that $g_x = g_y = 0$. As is common for these movements in quasi-static walking, we will assume our CoM height and yaw about the pelvis to be approximately constant ($R_{CoM,z}(t) = c, M_z(t) = 0$). Finally, since our CoM height is not moving, we are assumed to be counteracting gravity ($F_{leg,z}(t) = -g_z$).This allows the three equations to be rephrased as:

$$\boldsymbol{F_{leg,y}(t) = \frac{-M_x(t)}{R_{CoM,z}}} \tag{17}$$

$$\boldsymbol{F_{leg,x}(t) = \frac{M_y(t)}{R_{CoM,z}}} \tag{18}$$

$$\boldsymbol{\frac{F_{leg,x}(t)}{R_{CoM,x}(t)} = \frac{F_{leg,y}(t)}{R_{CoM,y}(t)}} \tag{19}$$

Which combines with the third equation to be:

$$\boldsymbol{R_{CoM,z} \cdot \left[\frac{M_y}{R_{CoM,x}(t)} + \frac{M_x}{R_{CoM,y}(t)}\right] = 0} \tag{20}$$

Since we need to account for a pre-existing angular momentum, we integrate to:

$$\boldsymbol{R_{CoM,z} \cdot \left[M_y \cdot \ln R_{CoM,x}(t) + M_x \cdot \ln R_{CoM,y}(t)\right] = c(t)} \tag{21}$$

Where $c(t)$ is the desired momentum against which to compensate.

The actual force and thus the actual moment that can be provided from our 7 degree of freedom leg kinematic chain is a kernel of the configuration space. This is a complicated calculation alone, but the abstractions and restrictions of the walking controller are also unknown. Thus we approximate the maximum value trace we would need to ensure safety. Given a maximum trace of this kernel in X and Y with fixed Z, we can find a line for which we can guarantee that any needed force is achievable:

$$\boldsymbol{M_y = M_{y,max} - \frac{M_{y,max}}{M_{x,max}} \cdot M_x} \tag{22}$$

Thus we can express the safe region of steps as:

$$\boldsymbol{M_y \cdot \ln R_{CoM,x}(t) + M_x \cdot \ln R_{CoM,y}(t) < \frac{c_{max}}{R_{CoM,z}}} \tag{23}$$

Depending on the actuatable moments, center of mass height, and desired momentum, the boundary for this region is:

$$\boldsymbol{R_{CoM,x} = \max_{M_x}\left[\alpha(M_x) \cdot {R_{CoM,y}}^{-k(M_x)}\right]} \tag{24}$$

$$\boldsymbol{\alpha = e^{\frac{c_{max}}{R_{CoM,z} \cdot M_{y,max}\left(1 - \frac{M_x}{M_{x,max}}\right)}}} \tag{25}$$

$$\boldsymbol{k = \frac{M_x}{M_{y,max}\left(1 - \frac{M_x}{M_{x,max}}\right)}} \tag{26}$$

This boundary typically forms a concave upward curve which goes to 0 in Y as X goes to infinity and goes to infinity in Y as X goes to 0. As such, again, a linear asymptote can be fit to the curve. This asymptote is in the space of $R_{CoM,x}$ and $R_{CoM,y}$ and so can be found experimentally on the robot.

This scheme of A* does not constrain the orientation of the final footstep. This can be corrected trivially. If the first goal step can be achieved with correct yaw without “toeing in”, then simply alter the yaw. If it cannot, take that step and then alter the yaw of the second step. Then take an additional step to correct the yaw of the first goal step.

3.2.3 Perception. Our perception algorithms are responsible for providing situation awareness information to our operator and for enhancing the autonomous capability of our robot. The breakdown of operator and robot perception responsibilities is listed in Table 4.

Table 4. Perception Responsibility Breakdown

Operator Assistance	Robot Autonomy
Data Compression	Closed-Loop Control
Data Representation	Mapping for Safe Motion Planning
	Monitoring Task Progress

Perception algorithms can reduce the environmental data representation by replacing raw sensor data with symbolic and geometric descriptions. This will provide the operator with situational awareness even in bandwidth constrained communications environments. They can also represent and visualize data in forms that assist an operator in remotely operating a task.

The robot can become more autonomous by perceiving its environment while executing an action to ensure that an action is resulting in the intended effect. It can also create representations of the environment to plan its actions. We believe that increased autonomy will also require robots to reliably monitor key task events, allowing them to autonomously assess their own effectiveness in performing a task that involves interaction with the environment.

Many of our 3D perception techniques rely heavily on the Point Cloud Library (PCL) that was originally developed by Willow Garage (Rusu & Cousins, 2011).

3.2.3.1 Localization – Point Cloud Registration. A critical closed-loop perception function in a mobile robot is localization. In order for a robot to follow a long term mobile plan accurately, the robot needs to measure its own motion through the world. Using only dead reckoning estimates such as inertial and kinematic sensing will produce drift in positioning over time due to integration of sensing errors. Visual localization allows the robot to perceive its environment to measure motion against the fixed world.

Our primary localization mechanism is through registration of incremental LIDAR point clouds against an accumulated map. We assemble the scans of our spinning LIDAR, and register each assembled cloud against the map. Using a full 360° cloud generates a more reliable iterative motion estimate by constraining the incremental transform with points sampled evenly from all directions relative to the robot.

However, the LIDAR’s position may change within the period of rotation of the assembled LIDAR, so we need a coarse estimate of motion to assemble the scans properly. We use a dead reckoning estimate from our kinematics and inertial sensors to generate this estimate, which is reliable for short time scales. To avoid map deformations or incorrect pose estimates that could occur from improperly assembled clouds, we spin our LIDAR at 5.0 rad/sec so that we receive fully assembled clouds at roughly 1.6 Hz. The trade-off is that each assembled cloud is sparser than if the LIDAR was spun at a slower rate, but we found that reliable estimates were generated by these sparse point clouds.

The point cloud registration process uses an Iterative Closest Point (ICP) algorithm. This incrementally minimizes an error function that aligns each incoming point cloud with the surfaces of a reconstructed point cloud map.

$$\boldsymbol{E} = \sum_{\boldsymbol{i}} [(\boldsymbol{R}\overrightarrow{\boldsymbol{p}_i} + \vec{\boldsymbol{t}} - \overrightarrow{\boldsymbol{q}_i}) \cdot \overrightarrow{\boldsymbol{n}_i}]^{\boldsymbol{2}} \tag{27}$$

For each point, each iteration selects the nearest neighbor in the point cloud map and calculates the projection of the separation between the two points onto the normal contained in the map at that point. We solve for a rotation and translation that can be applied uniformly across the new point cloud to minimize this error totaled over all points. This process is illustrated in Figure 42. We use the libpointmatcher C++ library developed by ETH Zurich Autonomous Systems Laboratory for the core iterative closest point (ICP) and nearest neighbor algorithm implementations (Pomerlau, Colas, Siegwart, & Magnenat, 2011).

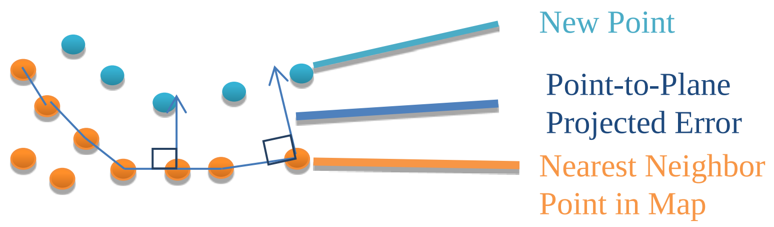


Figure 42. Iterative Closest Point Representation

To ensure that LIDAR point returns from the robot itself are not considered to be part of the static environment, we continuously filter points that we believe to be part of the robot based on our geometric and kinematic model in addition to our joint angle sensing.

To improve the quality of our map and filter the small amount of noise that our registration process can produce, we deactivate the ICP calculation process unless our robot is actively walking. Our dead reckoning estimate will only drift while the robot is walking.

3.2.3.2 Mapping. We build maps of the robot's environment to create a consistent 3D representation that accumulates knowledge from sensor data received from multiple points of view. We use the OctoMap framework from the University of Freiburg to represent our map as a probabilistic sparse octree. This structure avoids allocating memory for large empty regions of space. It also scales well to increasing environment size by creating a new root node at a lower level of resolution and re-parenting the old root node to it.

The process of updating the map from a point cloud source involves ray-casting from the sensor origin point to each sensed point in the cloud. We register the endpoint as a "hit" observation, and the points along the ray from the camera up to the end point as "misses." We give each sensor a model which defines its probability of hit and miss, and tune this according to its noise model. A typical OctoMap and population of an OctoMap via a sensor are shown in Figure 43.

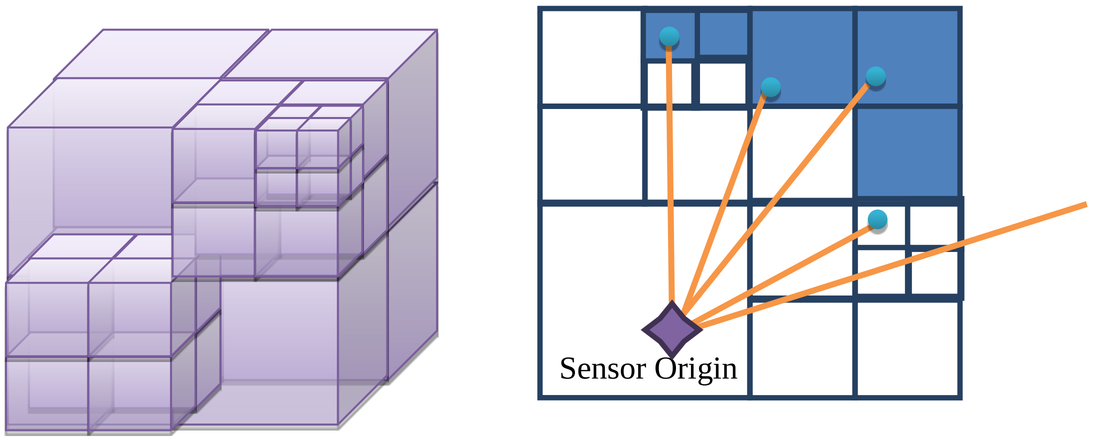


Figure 43. OctoMap Representation.

We store the log odds of occupancy in each node to speed up the update computation.

$$\boldsymbol{L(n|z_{1:t}) = L(n|z_{1:t-1}) + L(n|z_t)} \tag{28}$$

To further speed up this process, we only execute the ray-casting and free-space update step within 3 meters of the robot where manipulation requires that non-static objects to be updated over short time scales. The free-space processing step becomes computationally expensive at longer distances since the rays become longer and more accesses become necessary. We also remove points that we associate with the robot, the ground plane, and wall planes. This dramatically reduces the size of the map, and allows this data to be represented in a more efficient manner.

The data payload of our map contains not only occupancy, but also color and a timestamp of the most recent update. The color allows us to update the map with color observations within the view frustum of a color camera, but occupancy values only for LIDAR points that fall outside of any color information. The color value is integrated over multiple observations to smooth out any noise. When extracting occupied cells and rendering the map, we are able to use the color information for the operator's benefit. The timestamp data allows us to prune old data in our map, as the quality of data unobserved over a long timescale tends to be low. This is what allows us to avoid processing free space data at long distances because invalidated data will eventually be cleared out naturally over time. Figure 44 below shows the OctoMap representation of a valve and pipe after the walls and floor have been identified by plane detection and removed.

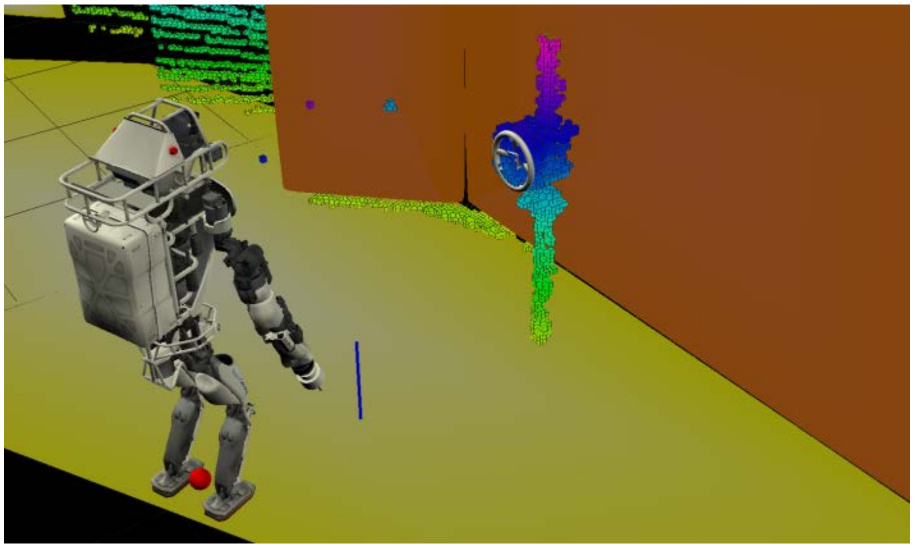


Figure 44. Atlas Maps a Valve

3.2.3.3 Plane Detection Our plane detection is based upon the PCL parallel plane and perpendicular plane sample consensus models. While the system defaults to using RANSAC, most of the other sample consensus algorithms are available as options. The plane detection module first segments its input cloud based on proximity through a k-d tree. For each segment, we run the parameter-selected sample consensus algorithm on the segment. If the sample consensus algorithm achieves a certain amount of inliers, we store its model parameters and convex hull before remove its inliers from the working cloud. (The working cloud is a copy of the initial input cloud.) Otherwise, we simply move on to the next segment. We store each pair of convex hull and associated parameters into a plane data structure and return an array of these structures and the remaining working cloud.

3.2.3.4 Shape/Object Detection Our object detection solution took the form of a hierarchical system of partitioning steps. The framework of the adaptive perception manager leads to pipelines of processing and recognition steps.

RANSAC is our preferred method for simple shape recognition. However, RANSAC is prone to false positives when there is a simpler shape in the environment that can encompass the convex hull of the desired shape. For example, a disc model will be placed on a low-error wall before it is fitted on a high error disc and a torus model will fit a disc before a torus. The easy solution is to detect the simpler shape and remove it from the actively considered cloud. RANSAC will also perform better on a more restricted cloud, thus filtering (spatial or otherwise) and segmentation (typically via k-d tree) will improve the end result. Each of these processes partition their input set into a set of outliers and at least one set of inliers. We found that the structure of each node of our partitioning tree was similar. A typical node is a pipeline of a filtering process, a segmenting process, and a more advanced process such as RANSAC. The inliers selected by the node are the inliers of the final process; the outliers are the relative complement of the inliers in the input set—the union of the outliers of each process. This flow is shown in Figure 45.

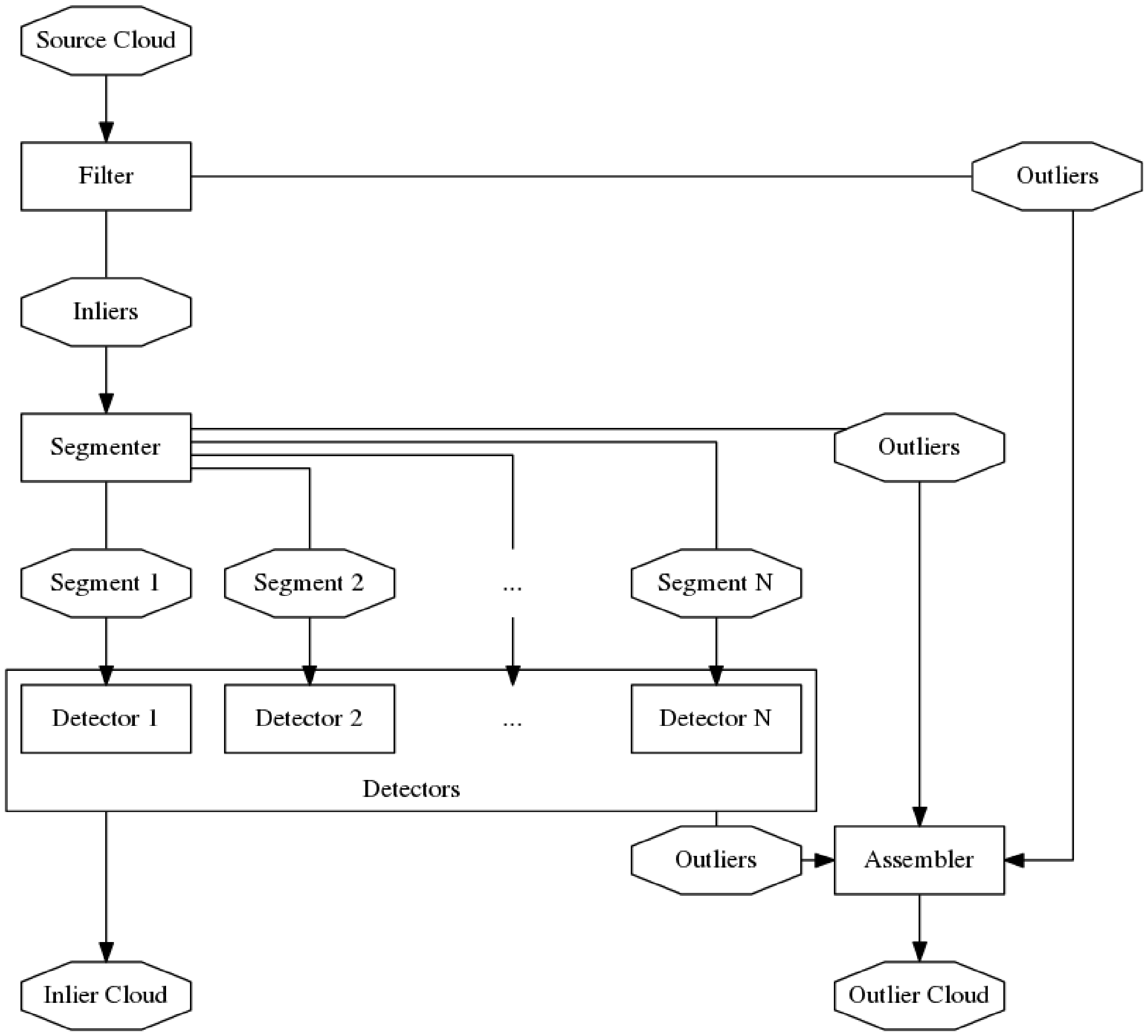


Figure 45. The Processing Flow for a Single Node of the Pipeline

These nodes can be connected so that uninteresting features are partitioned out of the pipeline in order of how easily and accurately they can be recognized. For instance, cropping to the 5cm above and below the floor is trivial if the robot has its feet on the floor and the transform from sensor frame to foot frame is available. This allows the floor plane to be detected quickly; then the next steps are done on the outlier cloud – the cloud with the floor removed. Each feature in the cloud is partitioned away until the only points in the actively considered cloud partition are either objects with which the robot must interact, obstacles to interacting with said objects, or dispersed noise points of a negligible number. Pruning the original cloud this way lets us more accurately detect these relatively unique objects. This hierarchical technique was employed for one of our door detectors, our valve detector, and our terrain field detector. The approximate pipeline for these three detectors is shown in Figure 46. Note that the door, valve, and terrain field detectors share processing steps removing the floors and walls; the adaptive perception manager guarantees that those shared steps are not duplicated in the system.

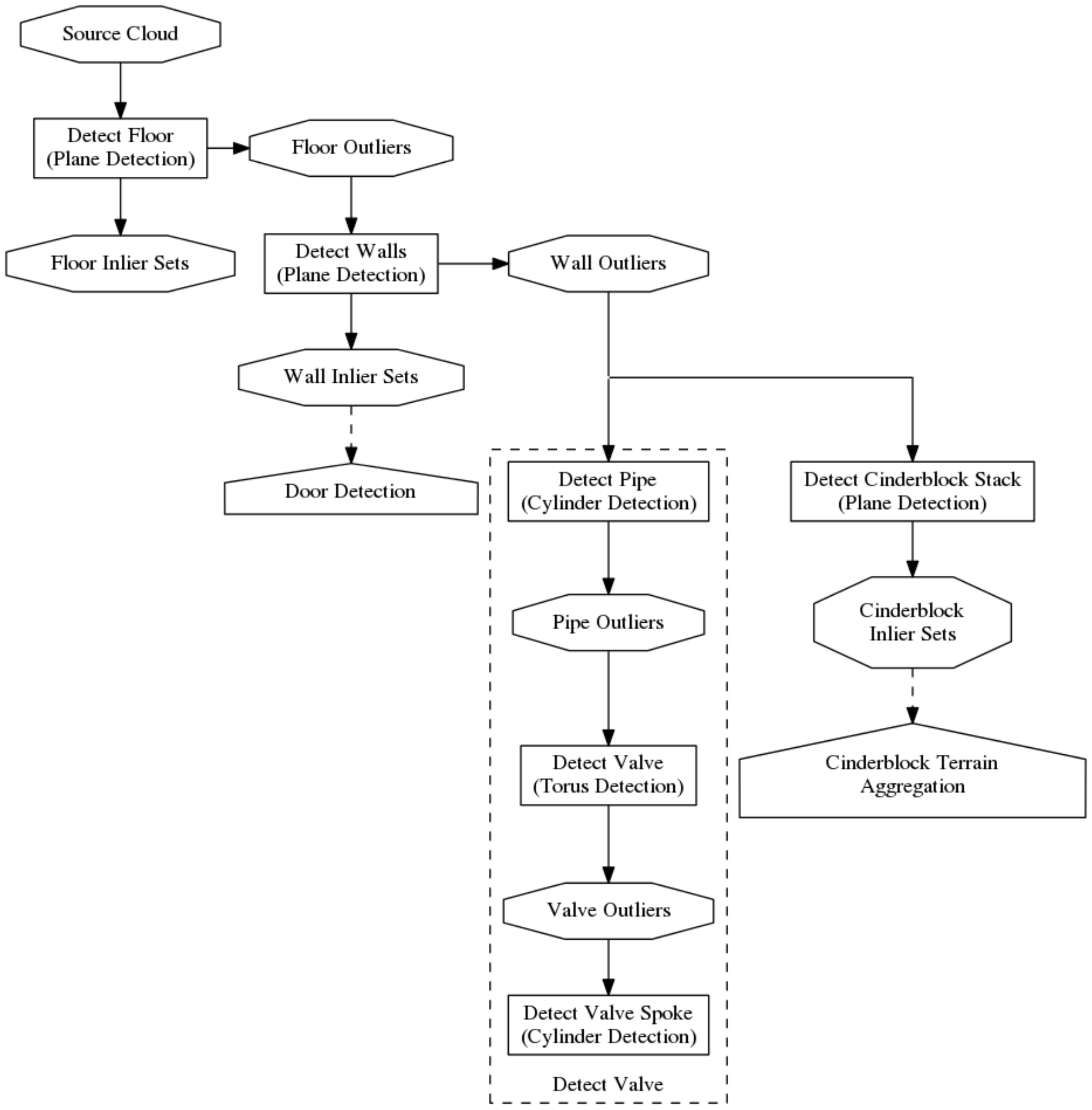

Source Cloud
Detect Floor (Plane Detection)
Floor Outliers
Floor Inlier Sets
Detect Walls (Plane Detection)
Wall Outliers
Wall Inlier Sets
Door Detection
Detect Pipe (Cylinder Detection)
Detect Cinderblock Stack (Plane Detection)
Pipe Outliers
Cinderblock Inlier Sets
Detect Valve (Torus Detection)
Cinderblock Terrain Aggregation
Valve Outliers
Detect Valve Spoke (Cylinder Detection)
Detect Valve

Figure 46. The Approximate Pipeline for the TROOPER System

3.2.3.5 Fiducial Tracking To provide more accurate end effector positioning for manipulation, we use vision to directly measure the end effector pose. This eliminates any error accumulation that exists in joint angle encoding throughout the kinematic chain. We use a fiducial detection library from NASA JPL that, using a stereo camera, can quickly search for and detect the 6-DOF pose of a specific fiducial marker given a pose that we seed through our kinematics and robot model. We have designed a wrist collar, pictured in Figure 47, which can be mounted between our robot forearm and end effector with repeated redundant markers that allow for detection from nearly all viewing angles. This approach decoupled the marker placement from the end effector itself, allowing us to use the same hand tracking configuration for arbitrary end effectors.

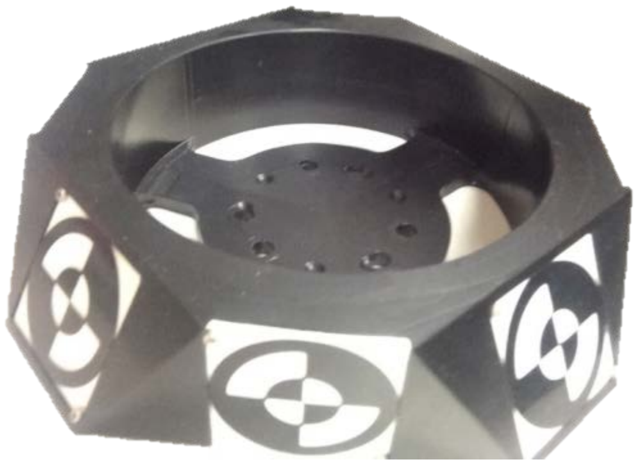


Figure 47. The TROOPER Fiducial Bangle

3.3 DRC Task Solutions

The DRC Finals were setup as shown in Figure 48. The course, from right to left, included:

A. Driving a Polaris Ranger
B. Exiting the Polaris Ranger
C. Opening and passing through a door
D. Locating and closing a valve
E. Picking up a drill and cutting through a wall
F. Surprise task
G. Rubble: clearing debris or walking over rough terrain
H. Climbing stairs

The robot was placed into the Polaris Ranger by the field team prior to starting a run. After driving and exiting the vehicle, the robot opened and passed through a non-spring-loaded push door. The robot then closed a valve by turning it at least one full revolution counterclockwise. Next, the robot picked up a cutting tool from a shelf and used it to cut out a black circle on a piece of drywall. The possible surprise tasks consisted of opening a small door and pushing a button, pulling an electric shutoff lever, or moving a power plug between sockets. The rubble task presented the option of passing through a debris field or walking over uneven cinderblocks. Lastly, the robot had to climb a series of stairs to reach a metal platform.

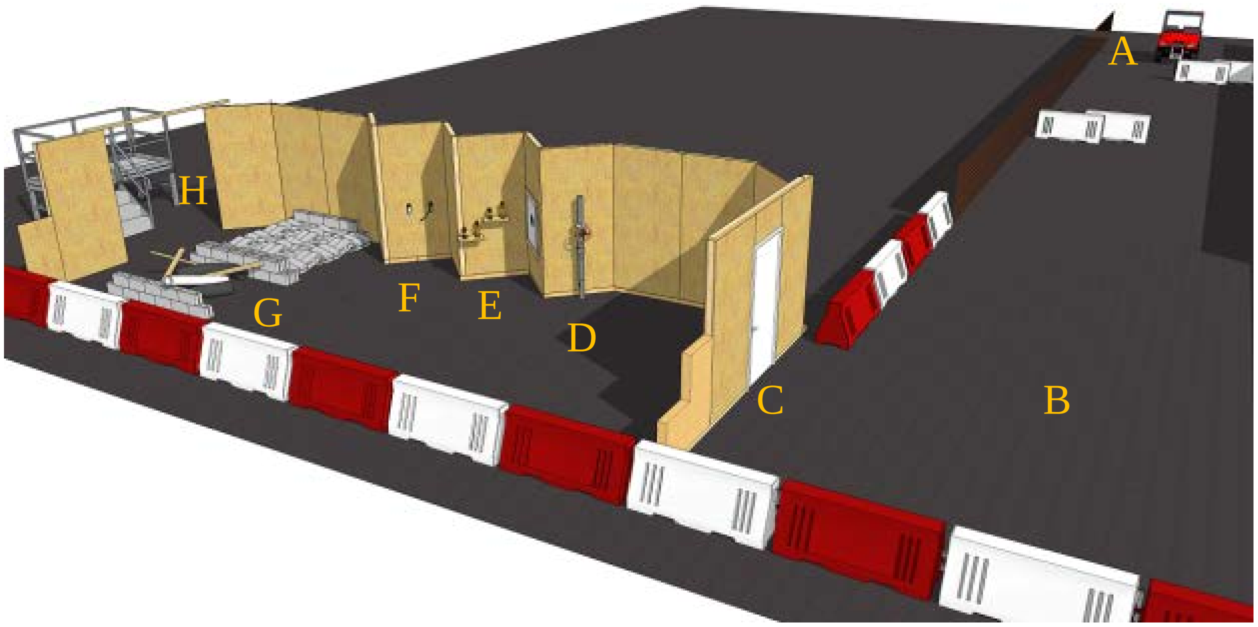


Figure 48. The DRC Course Layout

Practicality dictated solutions that were tailored to each task and worked around the limitations of the Atlas platform. For instance, our approach to driving and egress took into consideration the difficulty of getting Atlas to fit inside the vehicle. The approach for turning the valve used a radially symmetric end effector to allow wrist rotation. Actuating the drill was done using insets on the gripping hand, because positioning inaccuracy in the arms made pushing the button with a separate stick hand difficult.

3.3.1 Driving. The approach of team TROOPER for the driving tasks was partly shaped by the egress mechanism and partly by the overall size of the Atlas robot. After several tests, it was determined that the robot would have difficulty egressing the Polaris vehicle when sitting so that the Atlas was facing forward. Sitting the robot sideways in the vehicle would allow for less development time to be spent on the egress task as well as a higher success rate. The position of the robot for a sideways egress then directed the way in which the team pursued the driving task.

At the beginning of the run, the robot was positioned in a way such that it could manipulate its throttle and steering mechanism, described further in the following section. For the operator, an overhead map of the scene is presented. This map shows both the detected obstacles and the projected path of the vehicle given its current steering angle. Using the WASD commands shown in Figure 49, the operator is able to accelerate and steer the vehicle.

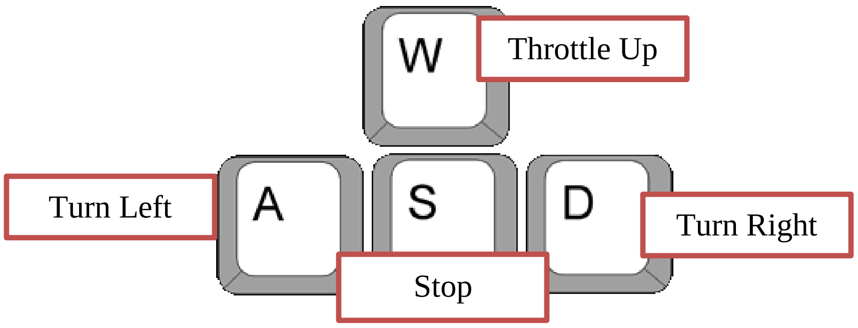


Figure 49. WASD Driving Commands

This configuration allows a comfortable interface that is common to many driving-based computer games. The general strategy for driving was slowly moving down the course. As the vehicle passed each obstacle, the operator would stop the vehicle, turn in the desired direction, and throttle up again. Throughout the task, the operator would view the overhead map in order to choose the steering direction. The process was then iterated on each turn until crossing the finish line.

3.3.1.1 Driving Mechanism. As described by DARPA competition rules, teams were allowed to use passive devices to aid in the driving of the Polaris Ranger. The stipulation was that this device must be installed within a five minute time window and must be installed without the use of tools. With these regulations in mind, the team developed a mechanism for driving the vehicle that independently operated the steering of the vehicle and the throttle. The left arm equipped with the Robotiq hand controlled the throttle while the right hand with the POKEY stick controlled the steering. First, the throttle mechanism (shown for the hand in Figure 50 and at the gas pedal in Figure 51) will be covered.

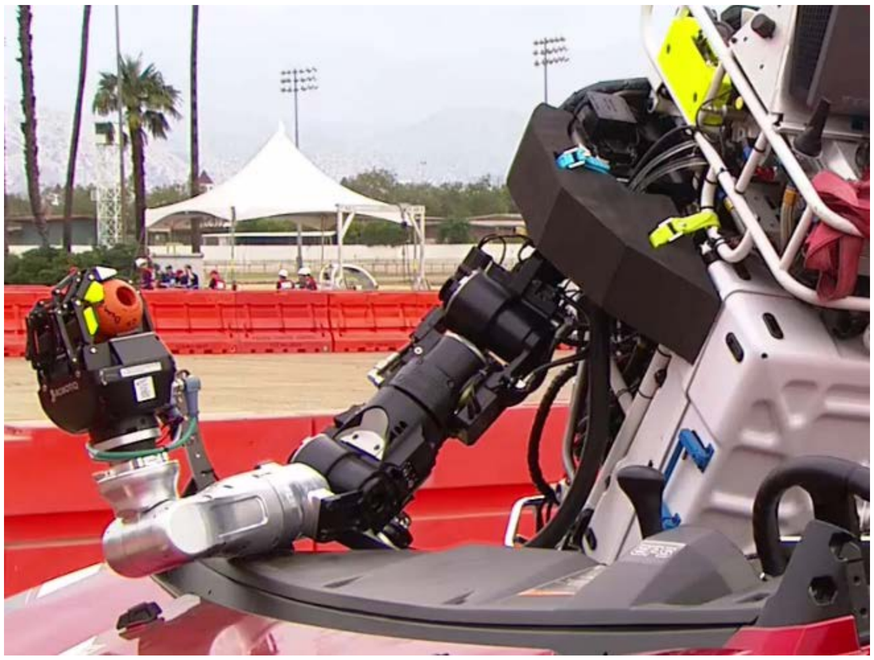


Figure 50. Throttle Control in Hand

After the robot was positioned to drive, a rubber compliant object was placed in the Robotiq hand and the hand was closed. Attached to the rubber object with a Bowden cable is a device to press the accelerator pedal in the Polaris. When the wrist turns, the Bowden cable is tensioned, which engages the pedal mechanism. The lever arm of the mechanism is positioned and shaped in such a way that twisting the wrist a small amount will result in the Polaris accelerating to a slow, desirable speed. Upon resetting the wrist, the spring on the pedal mechanism would restore the device to the disengaged position so that the vehicle would no longer move forward.

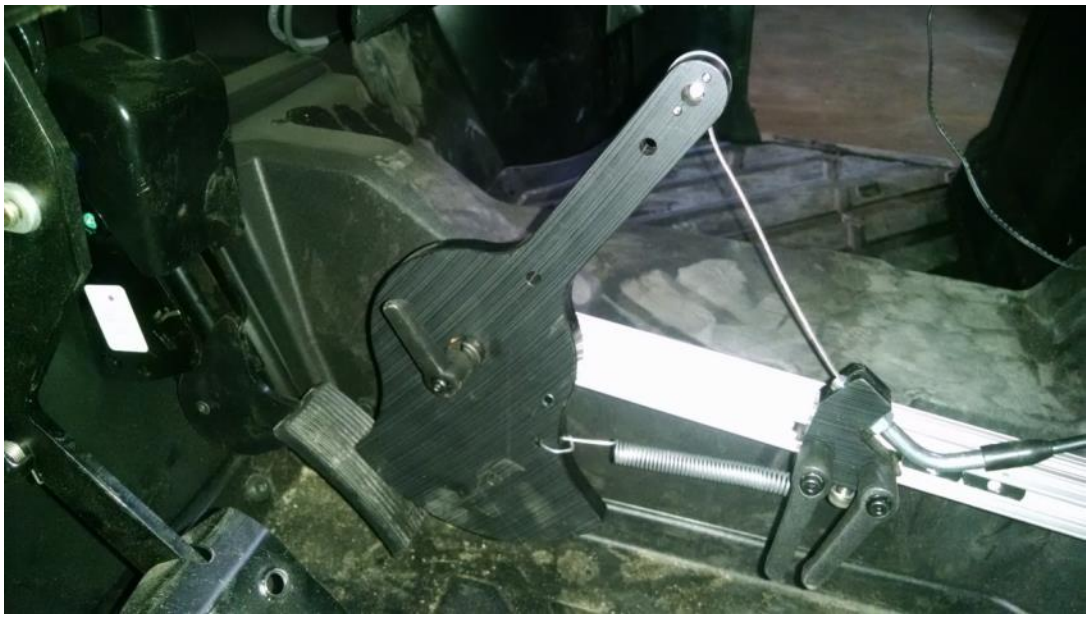


Figure 51. Pedal Mechanism

The steering of the Polaris is controlled using the right arm with the attached POKEY stick. The steering mechanism is made up of a chain and sprocket system, pictured in Figure 52. One side of the device is directly connected to the steering wheel and the other is connected to a device that fits around the end of the POKEY stick. The end of the POKEY stick is machined in a hexagonal shape that is complemented by the hexagonal hole in the steering mechanism. This works to keep the end effector snugly in position and move the connected steering wheel with little slip. When positioning the robot for driving, the wrist is put at its zero position and inserted into the steering hole. Using only the last joint on the wrist to control the steering has proven to give the range of motion necessary to complete the driving task.

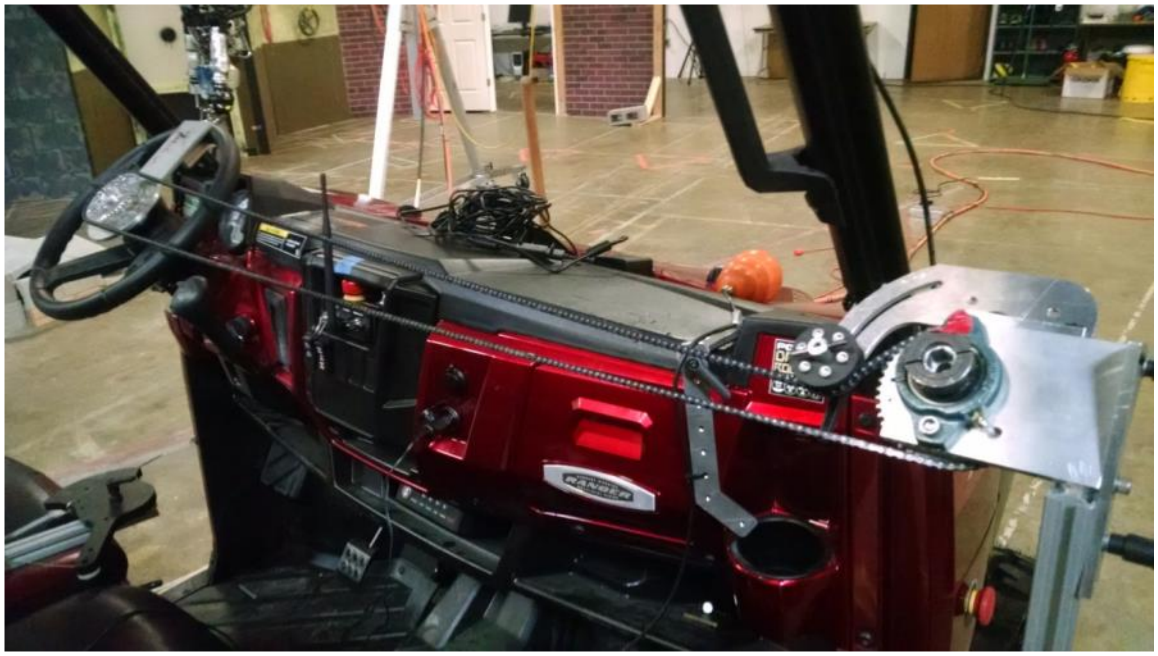


Figure 52. Steering Mechanism

3.3.1.2 Stereo Obstacle Detection and Mapping Due to the ATLAS' sidesaddle seated position within the vehicle, we were unable to use the MultiSense SL for obstacle detection while driving. Instead, we mounted a VI-Sensor, a stereo camera system with an onboard IMU, to the ATLAS as our main sensing unit while driving. By computing stereo disparities, we were able to obtain accurate depth information which allowed us to generate an overhead map of upcoming obstacles for the operator.

From the disparity image, computed using the ROS stereo image processing node, we performed ground plane detection with a modified RANSAC procedure. Between each frame, we maintained a global estimate of the ground plane by rotating our prior estimate according to the angular velocity measurements from the IMU. This estimate was then used during the random sampling stage by computing the angle between the normal of each triplet of points and the estimated normal and rejecting samples with angle greater than a fixed value. This procedure improved our inlier percentage significantly, reducing the number of RANSAC iterations needed and providing robustness against other, potentially larger, planes in the scene.

After we removed the ground plane, we filtered out noise by enforcing minimum blob sizes and temporal consistency for obstacles. We then converted the disparity map into a point cloud and used the RANSAC estimated ground normal to synthesize an image taken above the scene along the normal. On this image, we also plotted the instantaneous projected path of the vehicle given the current steering angle (read from the robot's wrist joint), using a simple bicycle model. The result can be seen in Figure 53 (right). In addition, we returned to the operator the left camera image with each non-zero disparity point colored by the type of point it represents, as shown in Figure 53 (left). The image shows a replica of the barriers used in the DRC, green points are ground, blue points are obstacles within the displayed range, and red points are obstacles above the displayed range.

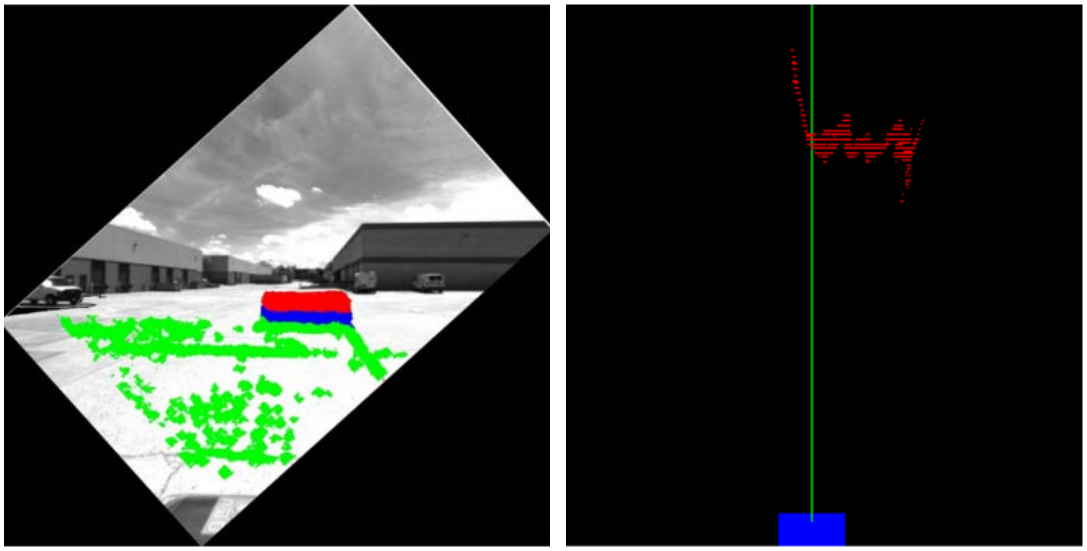


Figure 53. Stereo Obstacle Detection and Vehicle Model

Finally, we fused subsequent overhead maps together by tracking the vehicle's position using stereo visual odometry from the libviso2 package (Geiger, Ziegler, & Stiller, 2011). This fused map provided the operator information about obstacles all around the vehicle, rather than simply within the sensor's field of view. Unfortunately, we were unable to tune this method in time for the competition, but a sample fused map from a later sequence is shown in Figure 54. The left shows a color-highlighted image from a driving sequence on the highway; the center the corresponding overhead map; and the right a fused overhead map using libviso2.

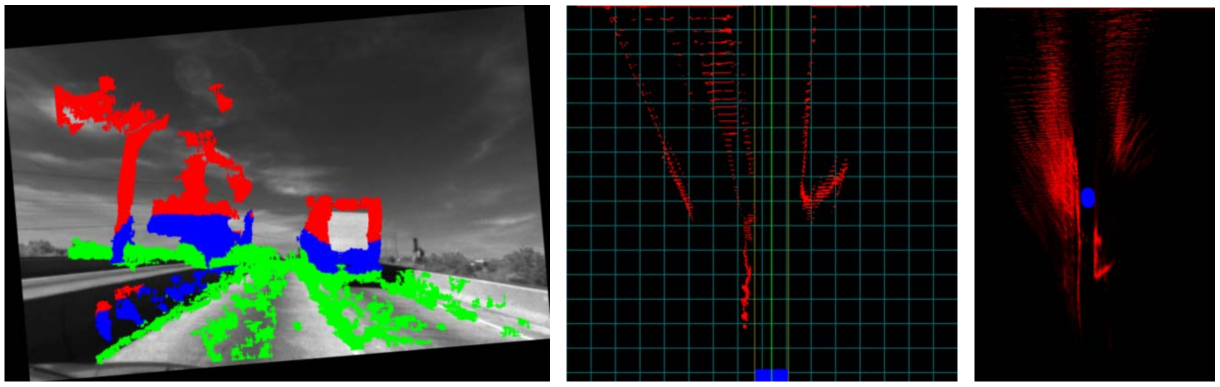


Figure 54. Highway Stereo Example

3.3.2 Vehicle Egress. For egress from the Polaris vehicle, a solution was conceived to rely heavily on a mechanical appliqué without much need for intricate software development and testing. By placing the robot into a series of tested and known poses, the robot sat up out of its driving position and began sliding out of the egress mechanism (as explained later). As the robot began sliding out, it assumed a stand-prep pose and gently touched the ground. The robot then raised its pelvis height to unlatch itself from the mechanism and took several steps away from the vehicle.

3.3.2.1 Egress Mechanism The mechanism for egress was focused around the robot having to move as little as possible in order to leave the vehicle. This inspired a design that allows for the robot to have one key frame for driving, then transition to a stand prep pose that would engage a linear slide to lower the robot to the ground. The mechanism is shown in Figure 55.

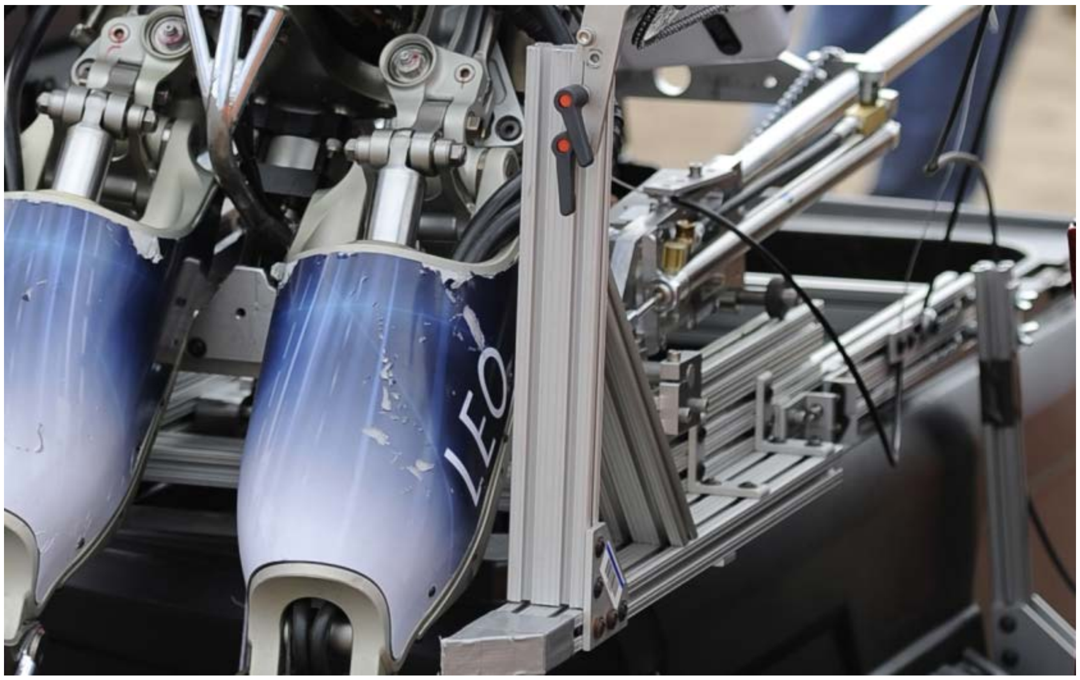


Figure 55. Driving Mechanism

During loading a series of quick releases are put into place, which lock both the slide and the robot into place. These remain secure throughout the driving event; upon completion of driving, the robot transitions into the stand prep key frame. At this point, the quick releases engage and the robot begins to slide down the mechanism. The slide mechanism was tested and iterated to ensure the device has the proper amount of throw to slowly and safely lower the robot to the ground. The slide itself has two degrees of freedom – one to slide out of the vehicle and one to lower the robot until its feet touched the ground. This setup allowed for a more compact and robust design than if just one degree of freedom was used. Each degree of freedom had its motion impeded by a hydraulic piston with a unidirectional flow restriction valve, allowing for a smooth motion to set the approximately 400 lb. robot down to a stable position on the ground.

3.3.2.2 Whole Body Egress Exit. The stages of vehicle egress are shown in Figure 56. In the left-most image, the robot begins the procedure by assuming a statically stable (stand-prep) pose. As the robot completes the motion the robot pulls the pin on the quick-release shackles that held the pelvis and seat anchored to the base. This is similar to disengaging a seatbelt. The robot’s weight begins to slide the robot forward. The slides are attached to hydraulic resistance to control the rate and prevent the robot from being thrust forward.

In the central image of Figure 56, the robot passes the pivot point and the slides begin to rotate. The rotation is slightly eccentric to allow the slides to clear the edge of the vehicle. Rotation uses hydraulic resistance to control the rate and prevent the robot from being rolled from the vehicle. Forks protruding from the seat to the robot waist also keep the robot aligned to the slide. Around 30 degrees the forks naturally disengage for upward and forward motion.

The slide and rotation are designed to allow for different vehicle angles including deflection of the vehicles off-road suspension system. In the right image, as the vehicle sheds the weight of the robot, its roll is absorbed by the slide which is now pitched 35 degrees upward. When the robot’s feet are squarely on the ground, the egress slide either naturally loses contact with the robot or can be nudged out of the way with some squats.

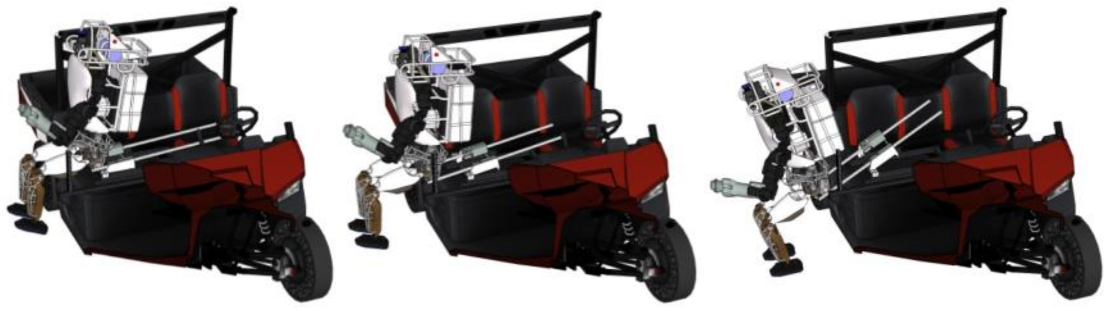


Figure 56. Atlas Vehicle Egress Progression

3.3.3 Door. The door task is the first task in which the principles of human-guided autonomy were applied to allow the operator and the robot to collaboratively complete a high-level goal. This task was initiated by the human specifying only that the robot should walk through a door and that it should use its left hand to open the door. The robot then provided the operator with a proposed task chain, such as the one pictured in Figure 57 below, which could accomplish this goal. Goals are shown in blue and tasks are shown in red. A sub-goal that is not colored indicates that it is a duplicate of a goal that appears elsewhere in the chain and it is expected that it will already be achieved by the time execution reaches that point in the chain. The numbers indicate the order of execution of the tasks. An explanation of each of these steps is given in Table 5. The door task chain includes 9 distinct tasks that make use of 5 different types of behaviors.

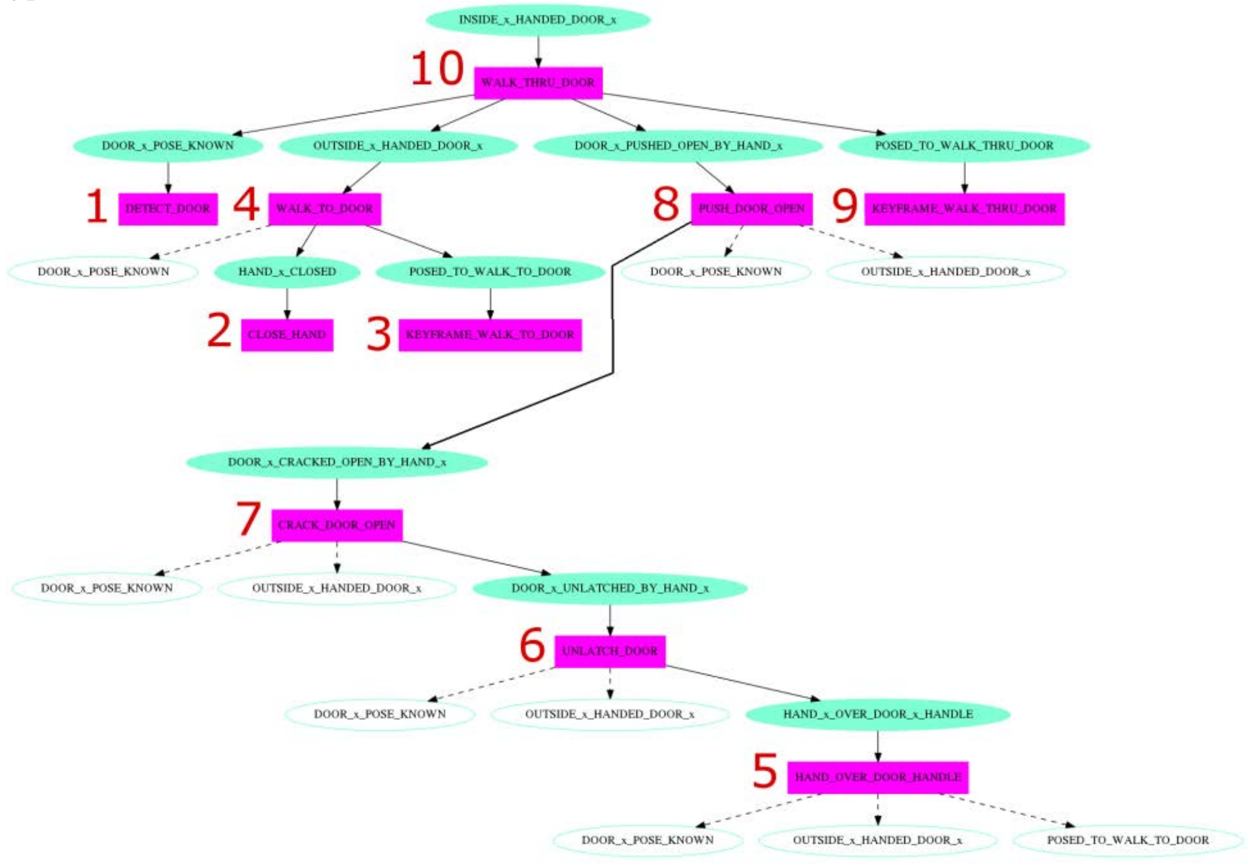


Figure 57. A Task Chain to Detect and Walk through a Door

Table 5. The Door Task Chain

Task	Behavior	Description
Detect Door	Detect Object	Find a door and add a corresponding door frame to the shared world model (detailed description provided in following section)
Close Hand	Control Hand	Close the left hand to prevent damage to the fingers when pushing on door handle
Keyframe – Walk to Door	Keyframe	Execute a pre-recorded set of joint positions to lift the arms above the robot's waist
Walk to Door	Walk To	Plan and execute a series of footsteps from robot's current location to just outside the door
Move Hand Over Door Handle	Task Space Move Hand	Plan and execute a collision free motion to move left hand from its current pose to just above the door handle
Unlatch Door	Task Space Move Hand	Plan and execute a motion to move left hand from its current pose to just below and beyond the door handle
Crack Door Open	Task Space Move Hand	Plan and execute a motion to move the left hand from its current pose to further inside of the door frame
Keyframe – Walk Thru Door	Keyframe	Execute a pre-recorded set of joint positions to hold the door open with the right hand and retract the left hand
Walk Thru Door	Walk To	Plan and execute a series of footsteps from the robot's current location to just inside of the door

3.3.3.1 Door Detection. We initially employed SAC plane model segmentation for the door detection task. The DRC door task was sufficiently constrained such that three discrete planar surfaces; a plain wall, the door itself, and surrounding frame, could be teased apart provided the segmentation distance function was parameterized to sufficiently discriminate between the depths of the surfaces (while not over-discriminating for finely-textured features on the surface of the door or wall themselves). The door itself is shown in Figure 58. As an initial step, a generalized perception module for plane detection was run so as to subtract the ground plane and remove extraneous features on the ground plane or above the course (similar to the preprocessing step taken in the case of cinder block field characterization).

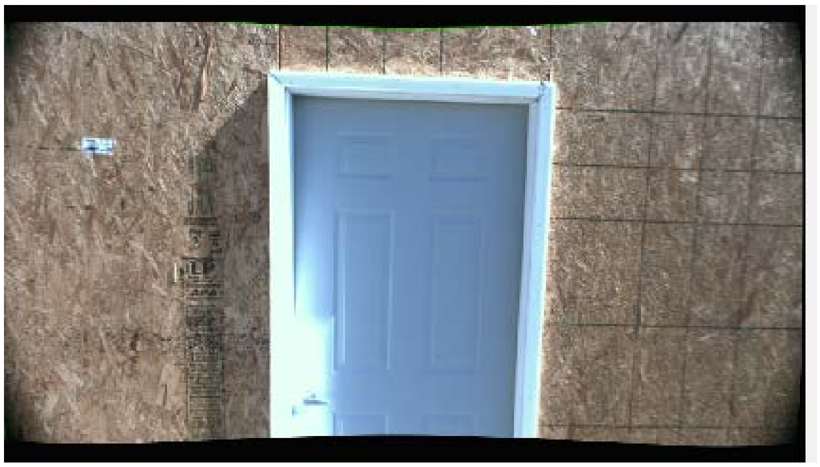


Figure 58. Charleston, SC Mockup of Door Task

While generally good at localizing the door plane, early experiments showed that despite the geometric regularity of the surfaces involved and the overall lack of clutter, SAC segmentation could not be guaranteed to provide the 2cm accuracy necessary for planning the unlatching movements of the arm and allowing the wide ATLAS platform to traverse the threshold. The principal reason for this has to do with the nature of the RANSAC algorithm. It is possible for RANSAC to converge on a plane model which is rotated slightly about the Z axis, resulting in a detection which is sufficiently imprecise to impede correct localization of the unlatching mechanism with odometry.

The solution to this problem was to eschew SAC segmentation in favor of exploiting the simple structure of the door frame and the reliable depth offsets. We developed a door detection perception module that uses LIDAR to find a consistent set of points set back from the rest of a wall surface. The module applies the following algorithm to detect the door.

1. Crop LIDAR point cloud at waistline
2. View frustum culling at 60 degrees
3. Project into XY-plane
4. Principle component analysis (PCA) to transform into Eigen-basis
5. Create threshold based on mid-point in minor dimension
6. Reorder points along major dimension
7. Increment through points, look for at least 1m of continuous points above the threshold
8. Use upper/lower bound transition points to determine the door position and yaw

The results from this door detection are shown in Figure 59 below.

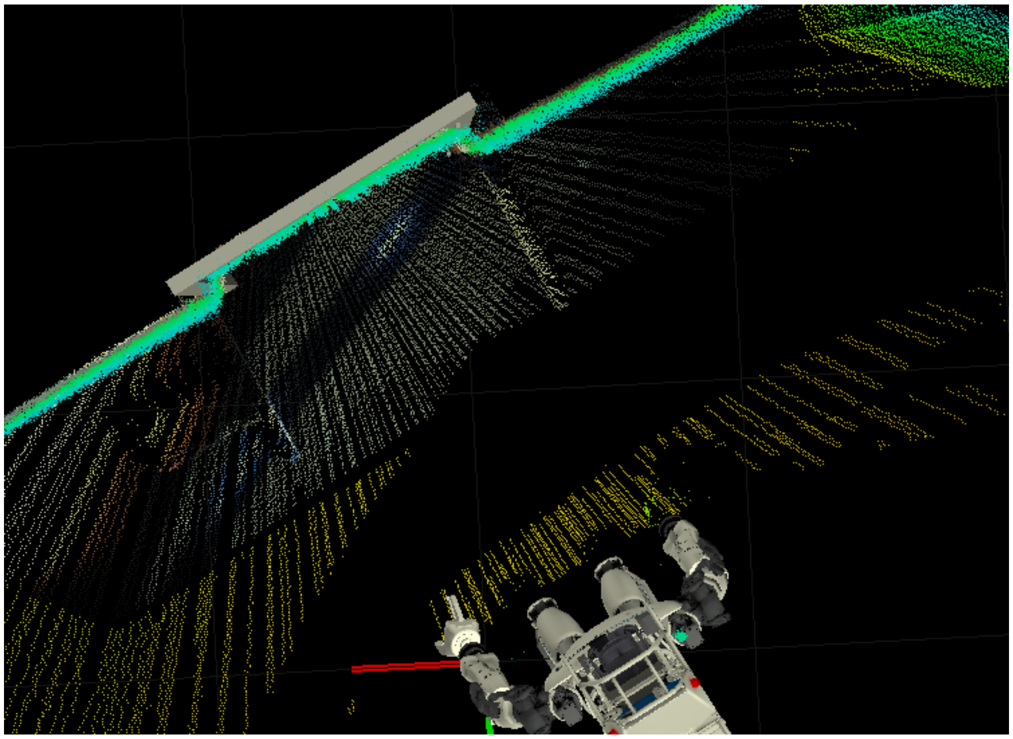


Figure 59. Atlas Autonomously Detects the Door Pose Using LIDAR Sensing

3.3.4 Valve. This task was initiated by the human specifying only that the robot should use its right hand to turn a valve. The robot then provided the operator with a proposed task chain, such as the one pictured below in Figure 60, which could accomplish this goal. Goals are shown in blue and tasks are shown in red. A sub-goal that is not colored indicates that it is a duplicate of a goal that appears elsewhere in the chain and it is expected that it will already be achieved by the time execution reaches that point in the chain. The numbers indicate the order of execution of the tasks. The steps in this chain are described in Table 6; the chain includes 9 distinct tasks that use 4 different types of behaviors.

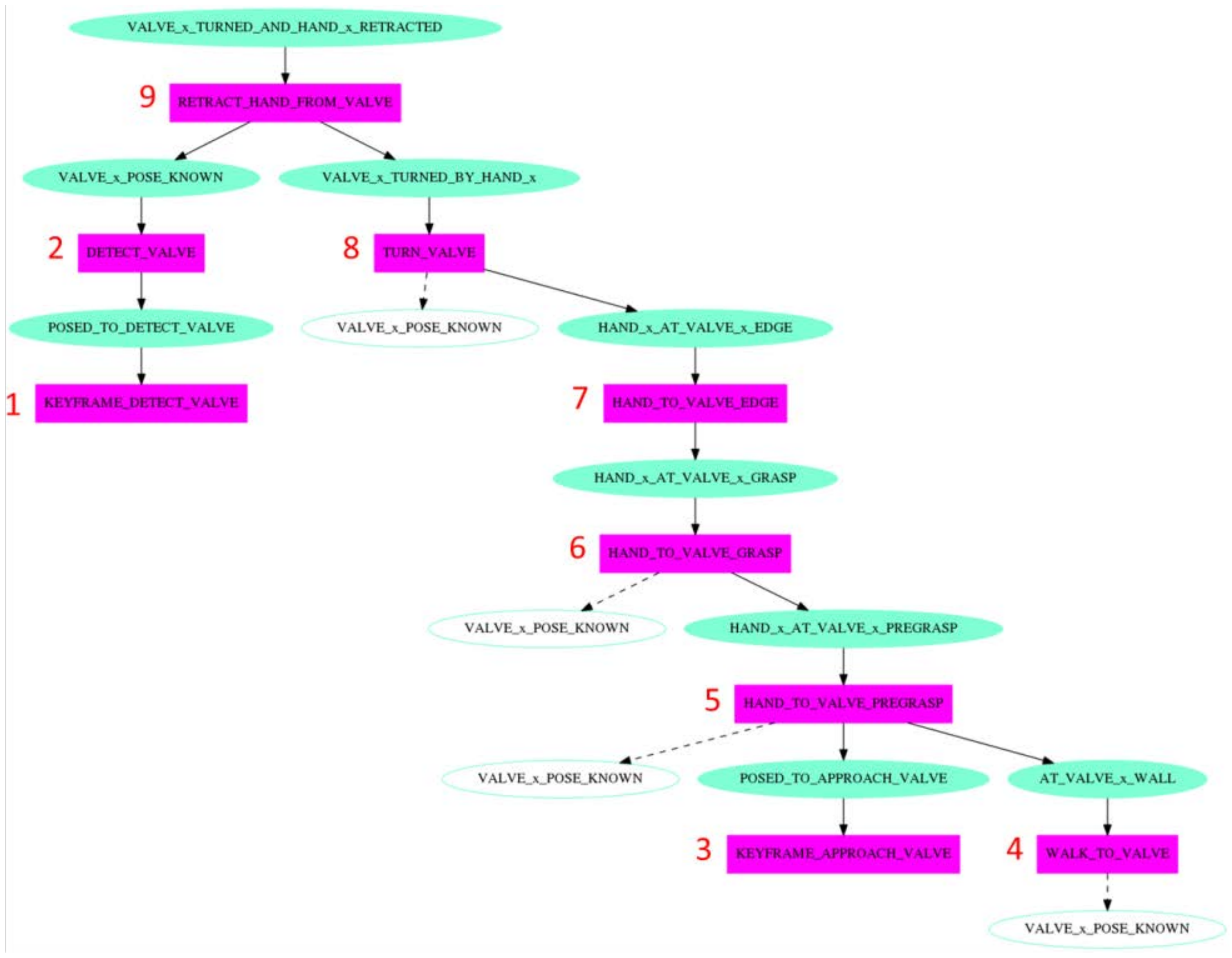


Figure 60. A Task Chain to Detect and Turn a Valve

Table 6. The Valve Task Chain

Task	Behavior	Description
Keyframe – Detect Valve	Keyframe	Execute a pre-recorded set of joint positions to move the arms out of the robot’s field of view
Detect Valve	Detect Object	Find a valve and add a corresponding object to the shared world model (detailed description provided in following section)
Keyframe – Approach Valve	Keyframe	Execute a pre-recorded set of joint positions to move the arms into a stable pose for walking
Walk to Valve	Walk To	Plan and execute a series of footsteps from robot’s current location to just in front of the valve
Move Hand To Valve Pre-Grasp	Task Space Move Hand	Plan and execute a collision free motion to move the POKEY stick from its current pose to the valve
Move Hand To Valve Grasp	Task Space Move Hand	Plan and execute a collision free motion to move the POKEY stick from its current pose to a small stand-off distance from the valve
Move Hand To Edge of Valve	Task Space Move Hand	Plan and execute a motion to move the POKEY stick from its current pose to the outer edge of the valve
Turn Valve	Task Space Move Hand	Plan and execute a circular trajectory for the POKEY stick (detailed description provided in later section)
Retract Hand From Valve	Task Space Move Hand	Plan and execute a motion to retract the POKEY stick from the valve

3.3.4.1 Valve Detection. Valve detection presented a difficult problem with existing infrastructure. Within our group, some had called it impossible. One major difficulty is that the major recognizable volume of a valve is either a flat ring or a torus. These volumes are concave and, worse, occupy a low percentage of the volume of their convex hull. The sample consensus answer to this situation is to also select based on the normal difference of each point from the model. Unfortunately, our vision pipeline sits upon compiled LIDAR scans—which did not produce especially accurate or useful normals.

Thus, we pursued a modified tactic. We created a PCL sample consensus model for tori that would actively detect the wheel of the valve as usual for a point-based sample consensus. This model detects as model coefficients: the radius to the middle of the wheel rim, radius from that middle to the edge of the rim, and the position and orientation in three dimensional space. To combat the false positives, we removed all anticipatable shapes that would produce false positives on the convex hulls of our target volume. In addition to removing all walls and the floor from the working cloud, we removed all vertical cylinders over a certain radius to exclude the pipe from the working cloud

Since a torus is symmetric around an axis, we also needed some determiner for the angle at which the valve was initially detected. The angle was intended to be used for avoiding the spokes of the valve, so we used PCL cylinder detection at a small max radius. The angle between the rejection of the cylinder axis from the torus axis and the rejection of the Z-axis from the torus axis is the angle for our valve about its axis.

3.3.4.2 Circular Trajectory for Turning Valve. In the case of turning a valve, we used the Drake IK with gaze constraints to constrain the stick to remain perpendicular to the valve face while allowing rotation about the stick axis. We used a circular series of poses and solved both on the UI and robot side of the communications divide so we didn't need to send the plan over a narrow communications channel.

3.3.5 Wall. This task was initiated by the human specifying only that the robot should use its left hand to pick up a cutting tool off of a particular shelf and use it to cut a particular pattern in a specified wall. The robot then provided the operator with a proposed task chain, such as the one pictured below in Figure 61, which could accomplish this goal. Goals are shown in blue and tasks are shown in red. A sub-goal that is not colored indicates that it is a duplicate of a goal that appears elsewhere in the chain and it is expected that it will already be achieved by the time execution reaches that point in the chain. The numbers indicate the order of execution of the tasks. The steps in this chain are described in Table 7, including 13 distinct tasks that make use of 6 different types of behaviors.

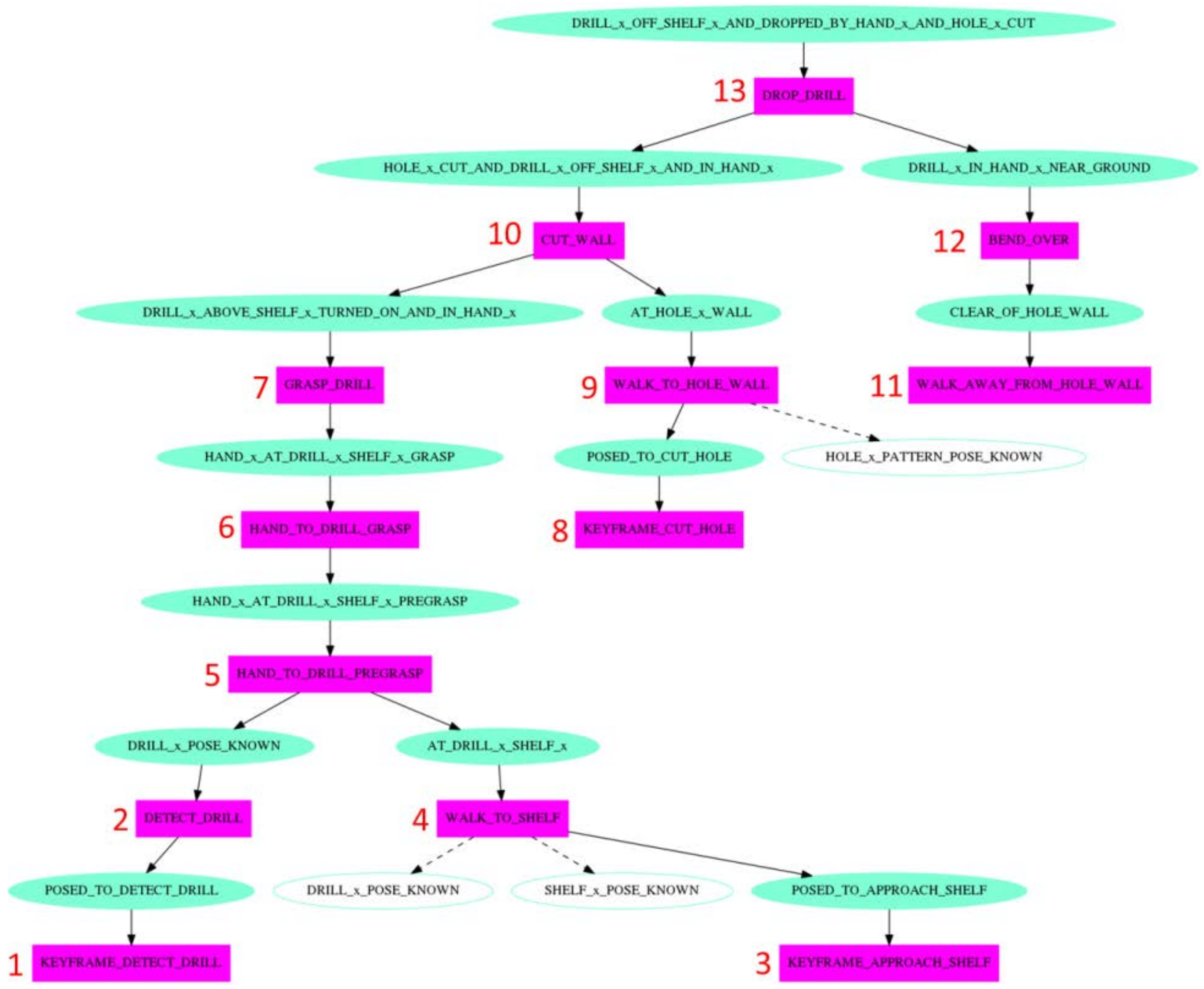


Figure 61. A Task Chain to Cut a Hole in a Wall

Table 7. The Wall Task Chain

Task	Behavior	Description
Keyframe – Detect Drill	Keyframe	Execute a pre-recorded set of joint positions to move the arms out of the robot's field of view
Detect Drill	Detect Object	Find a cutting tool on the specified shelf and add a corresponding object to the shared world model (detailed description provided below)
Keyframe – Approach Shelf	Keyframe	Execute a pre-recorded set of joint positions to move the arms into a stable pose for walking
Walk to Shelf	Walk To	Plan and execute a series of footsteps from robot's current location to just in front of the shelf
Move Hand To Drill Pre-Grasp	Task Space Move Hand	Plan and execute a collision free motion to move the left hand from its current pose to a small stand-off distance from the cutting tool
Move Hand To Drill Grasp	Task Space Move Hand	Plan and execute a motion to move the left hand from its current pose to the cutting tool
Grasp Drill	Grasp Drill	Close hand around cutting tool, lift the tool, and continue to cinch the tool until the tool is powered on (detailed description provided below)
Keyframe – Cut Hole	Keyframe	Execute a pre-recorded set of joint positions to move the left hand to a cutting pose
Walk To Wall	Walk To	Plan and execute a series of footsteps from robot's current location to just in front of the wall
Cut Wall	Task Space Move Hand	Plan and execute a circular trajectory for the left hand (detailed description provided below)
Walk Away from Wall	Walk To	Plan and execute a series of footsteps to walk backwards from the robot's current location
Bend Over	Task Space Move Hand	Plan and execute a motion to squat down and place the left hand near the ground
Drop Drill	Control Hand	Open the hand to allow the cutting tool to drop to the ground

3.3.5.1 Cutting Tool Detection. Our cutting tool detection module uses a colored stereo point cloud. We preprocess the point cloud to focus it on the height suitable for manipulation, roughly from the robot's pelvis to its head. Region growing segmentation with color separates the point cloud into chunks that are clustered spatially and have similar color features. We first down-sample the points using a voxel grid filter at 0.5 cm resolution to speed up the segmentation process. Figure 62 shows the drill detection in action. The left image shows the raw colored stereo cloud with a coordinate frame representing the detected tip frame of the cutting tool; the right image shows the voxelized cloud, colored by segment.

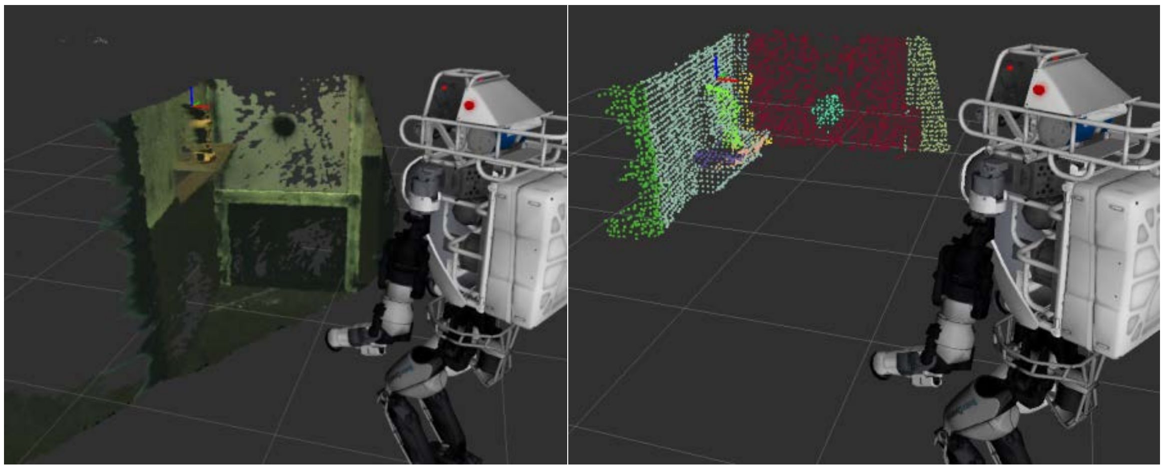


Figure 62. Cutting Tool Detection

Our detection process loops through the point cloud segments to find a strong match for a cutting tool. We require that the principal dimension of the cluster be roughly 28 cm, and the minimum dimension roughly 9 cm. We also require that the tool be standing vertically, so its angle must match the gravity vector with a small tolerance. This will not work if the tool falls flat on the table, in which case the segmentation algorithm would struggle to cluster the tool separately from the shelf. We would be forced to rely upon operator-driven drill detection in this scenario.

Both of the DRC cutting tools were primarily yellow, so our detector also prefers to select clusters that are primarily made up of yellow points. We represent the color yellow in YCrCb color space so that we are robust to illumination variation.

This module adds a cutting tool object to the world model to notify the rest of the system (along with the operator) of the position of the tool. The origin of the tool is at its tip. Small position errors, along with error in orientation about the principal axis, can easily be corrected by the operator in the 3D interactive scene. Errors in roll and pitch result in substantial operator burden.

3.3.5.2 Cutting Tool Grasp. We ended up encoding a small set of valid grasp configurations for the drill as a Task Space Region centered behind the drill handle. The most crucial part was to get the height just right such that the 3D printed pieces attached to the fingers would activate the drill.

3.3.5.3 Tool Activation. Our primary method for activating the cutting tool is to grip it in such a way that a 3D printed attachment is able to press the trigger. The Robotiq hand is opened and wrapped around the back of the cutting tool. Upon closing the hand, the 3D printed attachment makes contact with and depresses the button. At this point, the cinching behavior begins by opening the hand slightly and moving the hand up slightly. From here, the hand will open and close several times until the cutting tool is activated. This method has proven to be fairly robust in turning on the drill. The hand with attachment is visible in Figure 63.

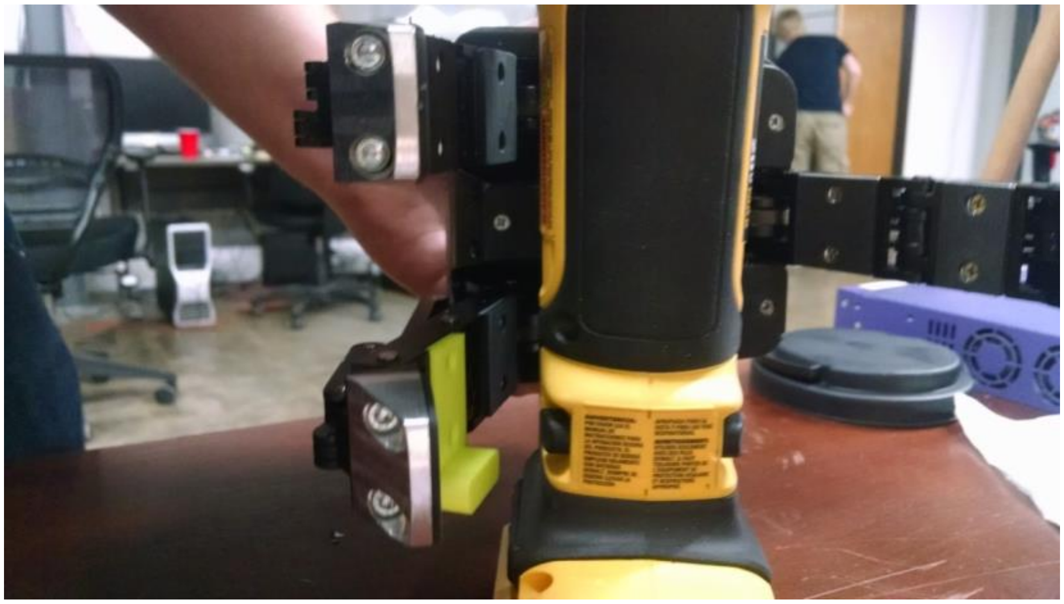


Figure 63. Drill Activation Tool

Our fallback method for activating the drill is to press the button by visual servoing in with the POKEY stick, described below.

3.3.5.4 Button Tracking and Stick Press. The task of visual identification and tracking of the drill button for manipulation and actuation is a well-constrained problem even under the varying lighting conditions at the Finals venue. It is well-suited to the popular mean-shift tracking algorithm. Mean-shift is an appearance-based tracking algorithm which uses histograms of pixel values, as in Figure 64, associated with a particular feature to track that feature across multiple frames. Histograms may make use of color features or more complex features such as line orientations and gradients. The algorithm functions as a non-parametric density estimator which generalizes to the problem of finding modes in a set of data samples.

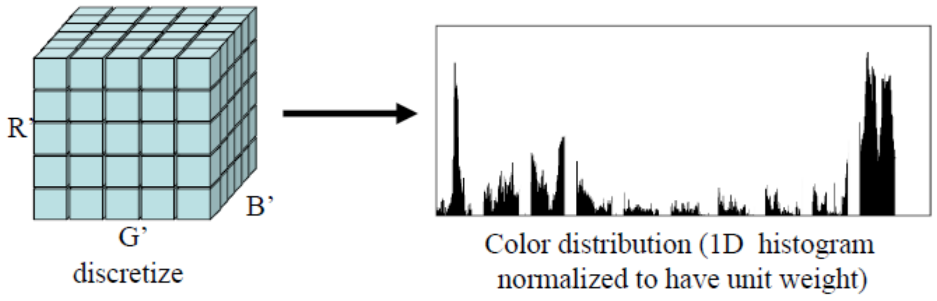

R'
B'
G'
discretize
Color distribution (1D histogram normalized to have unit weight)

Figure 64. Color Histograms for Mean-Shift Tracking

The power tool used in the DRC drill task has a conspicuous circular black button surrounded by the yellow chassis of the tool; this color combination is unique in the scene and the circular contours present a robustly trackable feature. Provided lighting variations do not exceed the dynamic range capability of the sensor mounted on the robot's hand, these features are also generally invariant to rotation. We use a variant of mean-shift, continuously-adaptive mean-shift (or CAMshift), which uses continuously adaptive probability distributions computed for each individual frame. This makes the algorithm more robust, but requires the use of spatial moments to climb the gradient rather than target and candidate distributions as in the conventional implementation.

3.3.5.5 Cutting Tool Activation Detection. Our robot and operator needed a method to determine whether the cutting tool was successfully activated before proceeding to cut the wall. We found that addition of a microphone to the POKEY-stick allowed the robot to monitor local sound intensity and detect a volume increase that corresponds with tool activation. In our testing, we found that providing the intensity data to the operator would enable them to recognize activation, even in the proximity of the active Atlas robot hydraulic pump.

3.3.5.6 Circular Trajectory for Cutting. The goal of the wall cutting task was to make a series of cuts such that the piece of drywall containing a black circle could be removed from the rest of the wall. Different teams made different shapes, though most could be categorized as either a large circle or square. We chose to go the circle route, and thus required something that could trace a large circle with the tip of the cutting tool. We used the Drake IK for this task. First, we created a circle to be traced in the hand frame and used the gaze constraints to constrain the cutting tool to remain perpendicular to wall while allowing rotation about the tool axis. Each successive IK pose to solve was seeded using the solution for the previous waypoint in the trajectory. Because a large number of waypoints were used, this planner ran on both the UI side and the robot side of the communications barrier; that way the trajectories were generated on both sides and did not have to be piped over a narrow communications channel.

3.3.6 Terrain. The terrain task was initiated by the human specifying only that the robot should walk to the other side of a cinder block field. The robot then provided the operator with a proposed task chain, such as the one pictured in Figure 65 below, which could accomplish this goal. Goals are shown in blue and tasks are shown in red. A sub-goal that is not colored indicates that it is a duplicate of a goal that appears elsewhere in the chain and it is expected that it will already be achieved by the time execution reaches that point in the chain. The numbers indicate the order of execution of the tasks. The steps in this chain are described in Table 8, including 13 distinct tasks that make use of 6 different types of behaviors.

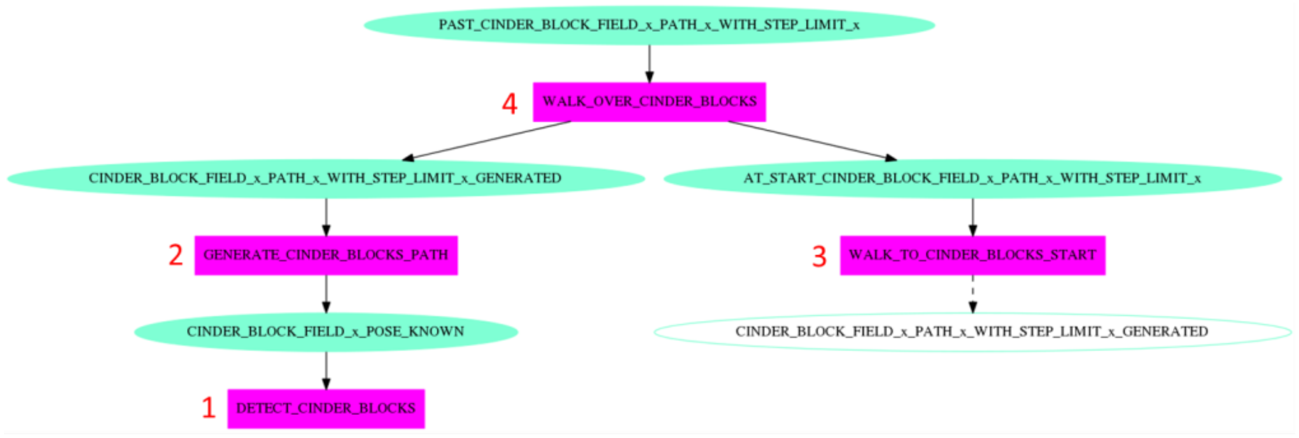

PAST_CINDER_BLOCK_FIELD_x_PATH_x_WITH_STEP_LIMIT_x
4
WALK_OVER_CINDER_BLOCKS
CINDER_BLOCK_FIELD_x_PATH_x_WITH_STEP_LIMIT_x_GENERATED
AT_START_CINDER_BLOCK_FIELD_x_PATH_x_WITH_STEP_LIMIT_x
2
GENERATE_CINDER_BLOCKS_PATH
3
WALK_TO_CINDER_BLOCKS_START
CINDER_BLOCK_FIELD_x_POSE_KNOWN
CINDER_BLOCK_FIELD_x_PATH_x_WITH_STEP_LIMIT_x_GENERATED
1
DETECT_CINDER_BLOCKS

Figure 65. A Task Chain to Walk over Cinder Blocks

Table 8. The Terrain Task Chain

Task	**Behavior**	**Description**
Detect Cinder Blocks	Detect Object	Find a cinder block field and add a corresponding object to the shared world model (detailed description provided below)
Generate Cinder Blocks Path	Walk To (planning only)	Use the description of the cinder block field to generate a footstep plan (detailed description provided below)
Walk to Cinder Blocks Start	Walk To	Plan and execute a series of footsteps from robot's current location to the location of the first two footsteps in the previously generated footstep plan
Walk Over Cinder Blocks	Teleop Walk	Execute the previously generated footstep plan to traverse the cinder blocks field

3.3.6.1 Terrain Field Detection. We represent the DRC terrain field as a rectangular grid of cinder block cells, as is fitting given that it appears in Figure 66. To find the terrain field, we use Euclidean distance clustering of a LIDAR point cloud focused at foot to knee height of the robot. We project each of the clusters into the X-Y plane and compute the convex hull of the points. We then iterate around the convex hull, searching for two long continuous edges separated by 180 degrees in orientation. We use these edges as the orientation of the field and compute the centroid of the cluster for its 2D position within the plane.

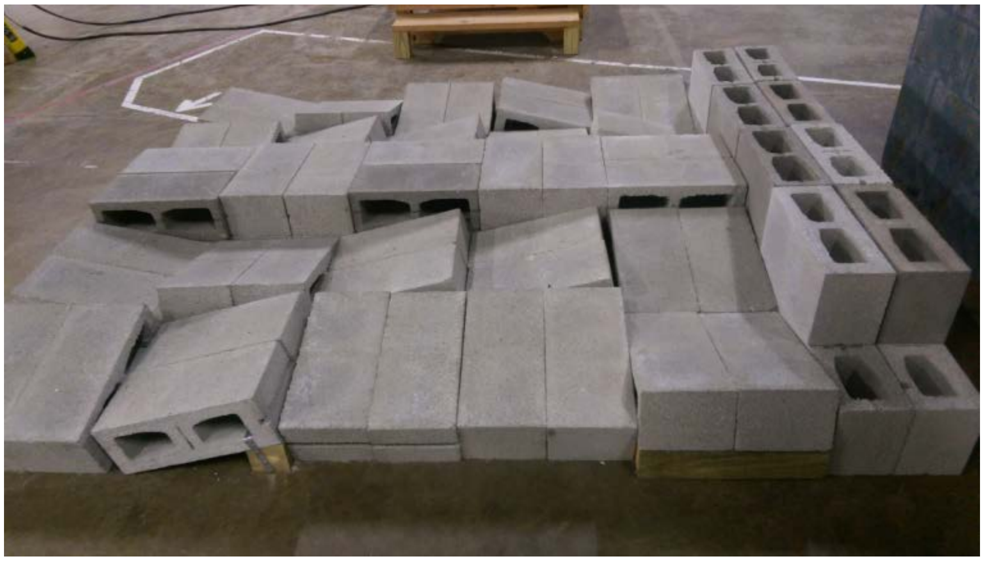


Figure 66. DRC Cinder Block Field as Recreated in Pennsauken Lab

3.3.6.2 Field Characterization. The geometric regularity of the DRC cinder block field traversal task admits some highly tailored ad hoc techniques for segmentation and characterization. Characterization for the purposes of path planning involves determining the height, incline, and extent of each discrete surface comprising the field in order to assist the robot with footstep placement.

The approach to characterization was predicated on DARPA's design for the field; a regular grid of cells, each consisting of the same surface area with a constant incline of 13º in one of four directions and at one of four discrete heights. In general, segmentation of planar surfaces at close range is easier with stereo disparity point cloud data. Experiments performed during the Charleston test event suggested that the dynamic range of the MultiSense SL head would prove problematic in the event of severe shadows as anticipated around midday in Pomona. The effects are shown in Figure 67, where half the field is not visible due to being in shadow.

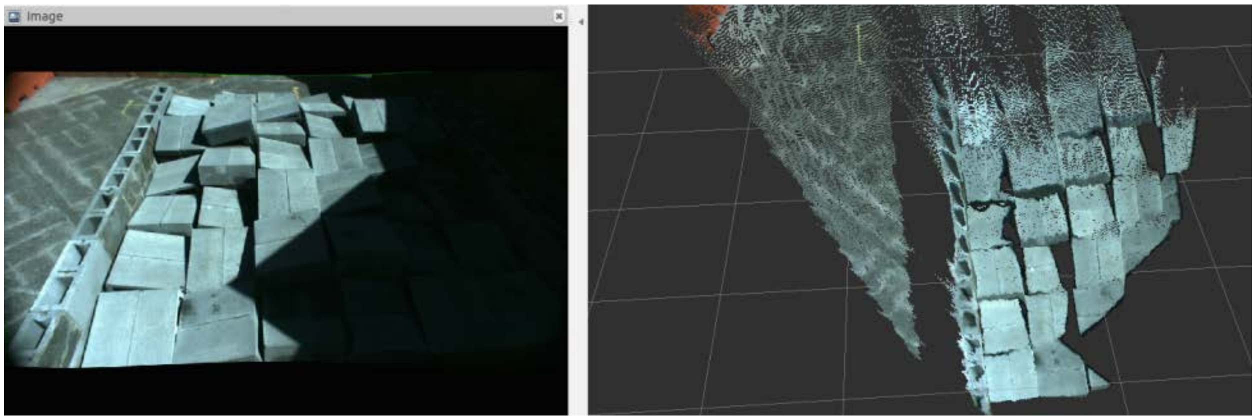


Figure 67. Effects of Dynamic Range Occlusion due to Midday Shadows

Segmentation of the cinder block field would then be performed primarily using LIDAR data which is more robust (albeit not entirely immune) to variations in outdoor lighting conditions. A Gazebo simulation LIDAR scan of the cinderblock field is shown in Figure 68.

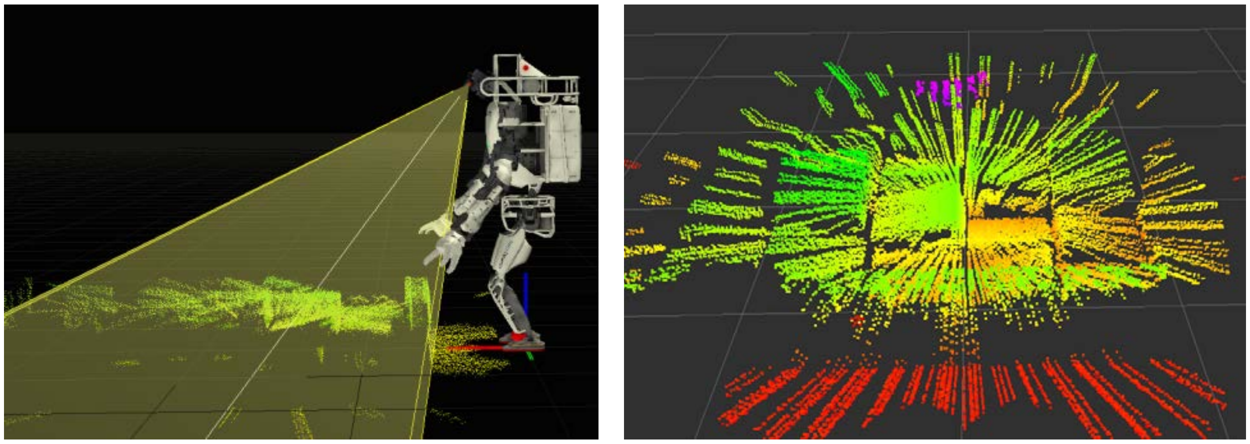


Figure 68. Simulated LIDAR Data from Cinder Block Field

Sampling and Consensus (SAC) model planar segmentation developed for the purposes of wall and ground plane subtraction is generally the preferred segmentation method; as implemented in the Point Cloud Library (PCL), it provides an automatic means to obtain the plane equation for each segmented surface and thus to determine the orientation of each discrete grid cell. SAC segmentation works by selecting an initial sample set of points, computing a model (planar model in this case), computing and counting inliers, and iterating until a specified maximum number of iterations or confidence threshold is achieved. An important point here is that the initial selection of sample points is random; thus, it is equally likely that SAC segmentation will result in a model being fitted to an area covering multiple planar surfaces at different orientations as to what the user identifies as a discrete planar object.

Early experiments with this segmentation showed that basic plane model segmentation with SAC as applied to the entire cinder block field would prove inappropriate for the particular task. While the cinder block field appears to us to be a highly regular sequence of highly discrete cells, SAC segmentation of the entire field is prone to try to fit surfaces to the entire field, a high-frequency jagged surface. In most cases, basic SAC plane model segmentation found characterized discrete surfaces only for the most elevated flat grid cells (on average, 2 or 3 per 6x7 grid), as visible in Figure 69.

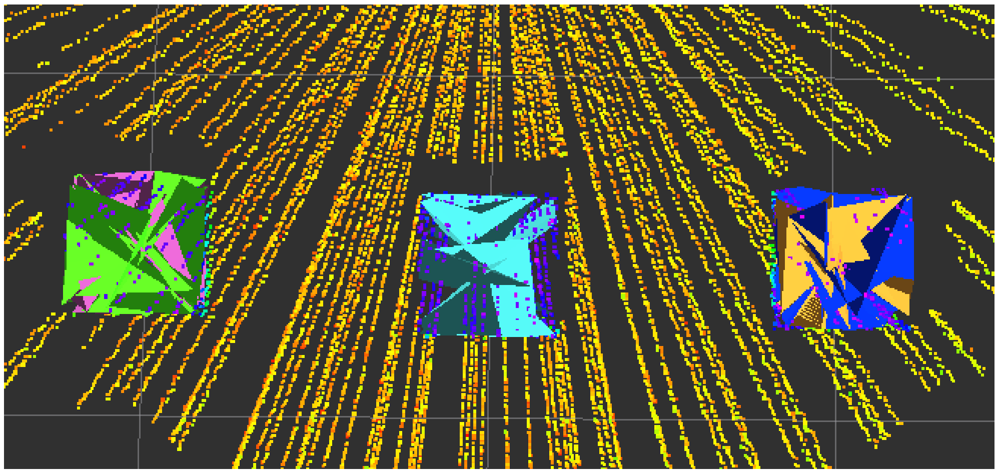


Figure 69. SAC Segmentation Results on Individual Cinderblocks (Simulated)

The solution to this problem was to apply a two-step segmentation process. Region growing segmentation is not subject to the same tendencies towards overfitting large planar surfaces to high frequency components, and was successfully used to segment individual grid cells. The algorithm then iterates over the segments found via region-growing and applies a SAC plane model fitting to each segment, yielding the geometry of the individual grid cells. Orientation is computed as the arctangent of the Z component of the surface normal, with the point cloud already transformed into the proper coordinate space. Figure 70 shows the output of this two-step process, the left being the results of region growing segmentation and the right the subsequent SAC plane fitting.

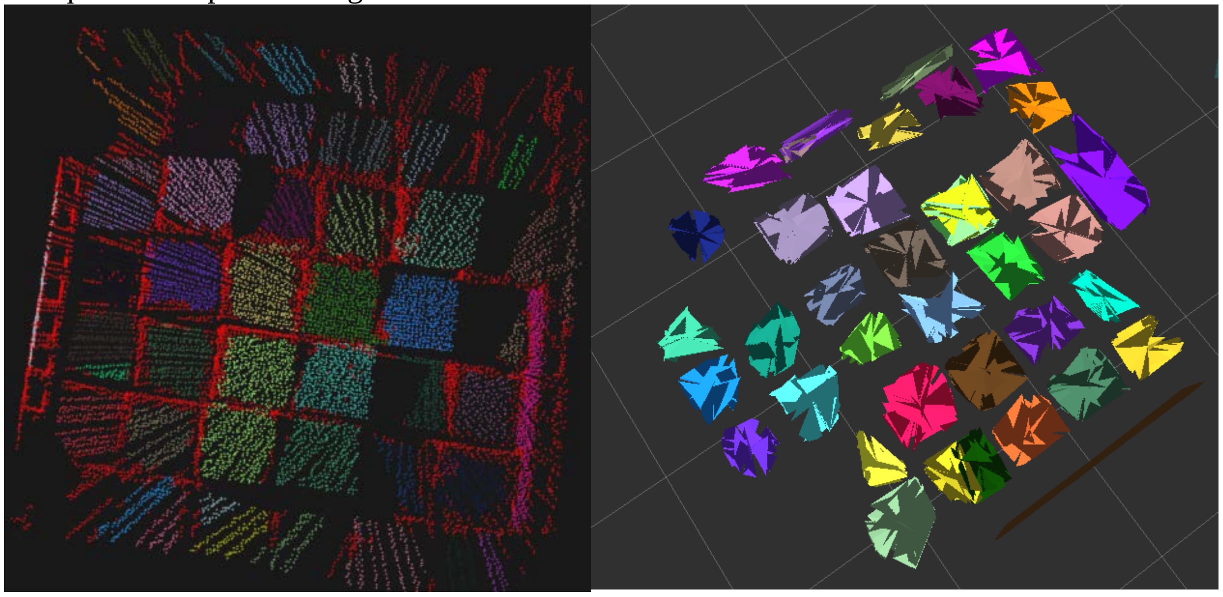


Figure 70. Cinderblock Field Detection

3.3.6.3 Footstep Planning. We solved the problem of planning footsteps over cinderblock terrain using an A* search with a set of allowed transitions informed by experimentation and advice from IHMC. The 4D state space consisted of the location of the left and right foot (left foot XY, right foot XY). The cinderblock field itself was discretized by each half (split left/right) stack of cinderblocks, so a 6x6 field of cinderblocks would be represented by 6x12 possible locations for either foot. Each transition consisted of a stance foot (the foot remaining stationary) and a swing foot moving to a new location. The cost assigned to each transition encoded the difference in height between the stance foot and the final step location.

We used the following transition rules found through experimentation and advice from IHMC to generate the underlying graph:

- Allowed table of transitions include stepping forward one cinderblock and stepping one half cinderblock to the side.
- Disallowed transitions that left stance foot on downslope when stepping down cinderblocks
- Disallowed any changes in step height of over a one cinder block height
- Disallowed leaving a stance foot on an upslope because we would hit the ankle torque limits

A sample path found by the planner is shown in Figure 71.

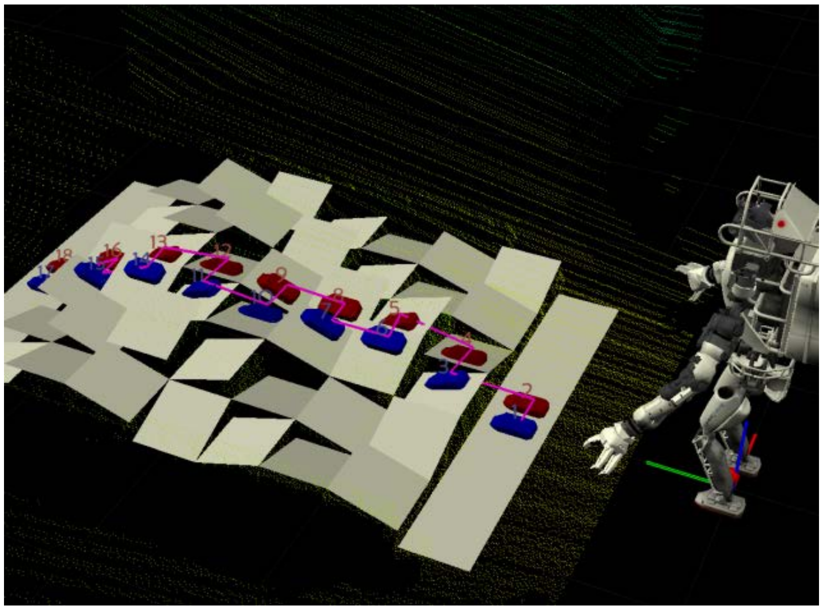


Figure 71. A Simulated Path over the Field

In practice, we also attempt to follow the seam between cinderblock columns as that was more stable than staying completely on one column. Straddling the seam allowed for more separation between feet as opposed to squeezing both feet onto one block. Figure 72 highlights the importance of not leaving a stance foot on an upslope when stepping down. Atlas is hitting a joint limit on his left foot, which causes him to slide on the blocks and could have easily resulted in a fall.

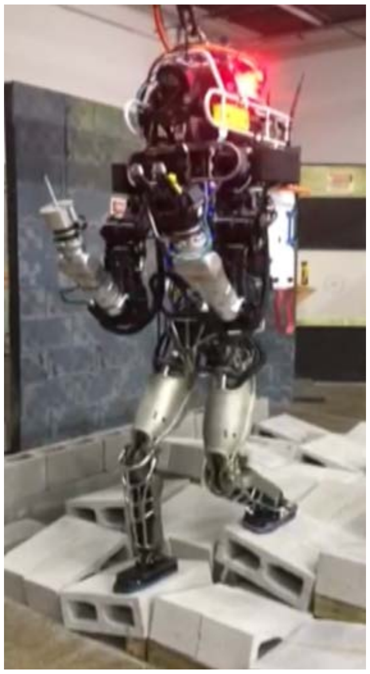


Figure 72. Atlas at a Joint Limit when Descending Cinderblocks

3.3.7 Stairs. Atlas faced a couple of issues when walking up the stairs. The stairs themselves were very close to the physical limits of what Atlas could traverse, specifically:

- Stair depth comparable to the foot length
- Stairs overhang catches toe when raising foot
- Shins collide with the next step when transitioning weight

The solution was alternating between half and full steps to surmount the stairs, all the while keeping the pelvis height at the maximum possible value. Only placing the front half of the foot on one stair allows enough room such that the shin does not collide with the beginning of the next stair when transferring the weight to the other foot. This said, only using half the foot required changes to IHMC's balancing algorithms; they exposed a method to allow specifying contact points that cover a limited subset of the foot. Figure 73 shows Atlas at the stairs. The left image shows a narrow stance with the pelvis raised, and the right image shows Atlas placing half its right foot on the first step. The right shin touches the next step, but is still within recoverable limits.

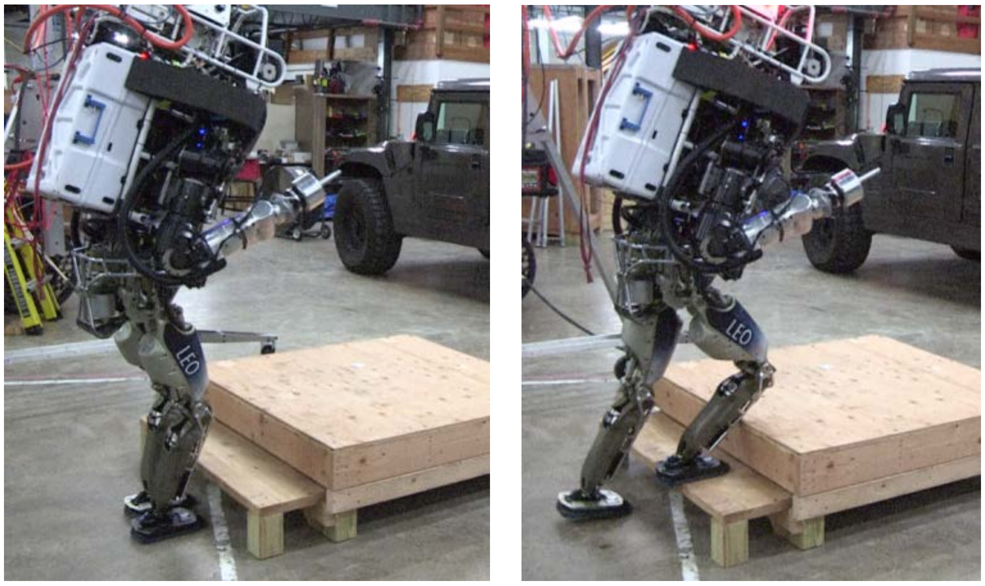


Figure 73. Atlas at the Stairs

We noticed other teams had different approaches; i.e., we saw that the Worcester Polytechnic Institute (WPI) team walked bowlegged, placing one foot at an outward yaw of approximately 45 degrees. This allowed them to place the whole foot on the step while still not hitting the shin.

4 RESULTS AND DISCUSSION

4.1 Approach to the Competition

As the competition approached, we knew we were one of the smaller teams competing. We debated internally as to which events we should focus on, possibly to the exclusion of others. We considered skipping driving, egress, and the wall task. In the end, we determined to drive and egress, but only attempt the wall task if we had extra time.

In this section, we detail the differences between the approaches written in previous sections and the approaches we decided to use in the competition.

4.1.1 Driving. Driving was carried out as described in Section 3.3.1. Much of the difficulty of the task was in properly installing Atlas inside the Polaris. The gas pedal had to be properly calibrated such that the vehicle RPM were in a certain range when the throttle mechanism was at maximum tension. The steering had to be properly centered as well. The operator steered using imagery from the stereo camera and a top-down view showing the anticipated vehicle curve given the steering angle and obstacles found using the stereo disparities. The camera gave a narrow view of the vehicle, making it difficult to judge the vehicle extents; more practice would have helped to alleviate this.

4.1.2 Egress. We carried out egress using the mechanism described in Section 3.3.2. We had planned to trigger the release mechanism for the slide using a solenoid attached to the pelvis, but testing showed that friction often caused this to fail. Instead, we used a series of quick releases that would trigger when the robot shifted its configuration inside the vehicle. Most initial tests ended in failure, with the robot falling sideways off the slide or getting caught on the slide when trying to walk away. It was only on the last day before the competition that we found a reliable, repeatable setup.

4.1.3 Door. We planned to use the task chain as described in the Section 3.3.3. In practice, we switched to teleoperation to open the door handle. We were able to unlatch the door with the operator in-the-loop. The locations for where to stand to unlatch and to move through the door proved very useful. We were able to walk forward through the door as long as the operator remembered to narrow the footstep width parameter in the user interface.

4.1.4 Valve. We planned to use a truncated version on the task chain described in Section 3.3.4 for the valve. The truncated chain would have included detecting the valve, deciding where to stand, entering an appropriate keyframe, and walking to the valve. The rotation of the valve would have been entered manually by the user as a circular end effector trajectory. In practice, we did not use the valve detection and instead relied upon teleoperation for the entire valve task.

4.1.5 Wall. We decided to skip this task in the competition. We had practiced picking up the drill and turning it on, but had not successfully cut the hole out of the drywall. With our limited manpower and time, we were unable to devote the necessary resources to this task.

4.1.6 Mystery Task. We did not practice the mystery tasks and resolved to teleoperate them in the competition.

4.1.7 Terrain. We decided to use teleoperation and manual footstep placement to cross the cinderblock terrain. We tested using the results of the A* terrain footstep planner described in Section 3.3.6 and found that the foot placements were slightly too far apart and that localization drift prevented the robot from traversing the entire field at once. Moreover, we learned simple heuristics from IHMC regarding foot placement, but decided the time it would take to translate them into a working planner was better spent on other critical issues. Additionally, in practice, we often encountered situations requiring intervention, such as needing to approach the edge of a cinderblock first before stepping down.

4.1.8 Stairs. We decided to use teleoperation and manual footstep placement to climb the stairs. We had debated about using the cinderblock planner to generate a trajectory for the stairs, but ultimately decided against it. We could have used a hard-coded footstep trajectory since the stair measurements were known, but our practice runs showed that it was easier and more reliable for the operator to place the steps. We used the strategy of alternating half steps and full steps to avoid hitting the robot's shins on the next step.

4.2 Competition Results

Our final results proved disappointing. Technical difficulties, hardware failures, and a lack of practice with our own system prevented us from progressing far into the interior course in either of our two runs. In the end, we completed the driving, egress, and door tasks and attempted the valve. Both of our runs resulted in 2 points being scored before we ran out of time.

4.2.1 Driving. On our first run, throttle was not properly calibrated, so when the robot attempted fully open the throttle it was not sufficient to move the vehicle. This led to our first reset. During the reset, the wireless network then went down course-wide and we lost our connection to the onboard computers. Afterward, we managed to successfully reach the goal area. Our stereo object detection and mapping proved poorly tuned for the environment and we found that our view through the VI sensor was not sufficient to gain an accurate idea of how much clearance was available for the Polaris. Consequently, we drove along the left edge of the track until the friction stalled the Ranger. Once the Ranger stalled, our issue became apparent and we were able to drive it away from the edge and complete the course.

On our second run, our now-functional stereo object detection and mapping allowed us to handily drive the course. However, halfway through the course, our driving mechanism detached from the Ranger unexpectedly forcing a reset. The detachment was not apparent from the operator interface, causing lost time. Once reset at the starting line, we attempted to drive forward, pulling increasingly harder on the throttle, while the vehicle remained stationary. The vehicle was not properly in gear; once the officials put it in gear we drove the course successfully, but it is likely this event caused one of the joints in our left wrist to become inoperable. Despite the broken left wrist, we reached the driving goal, shown in Figure 74.

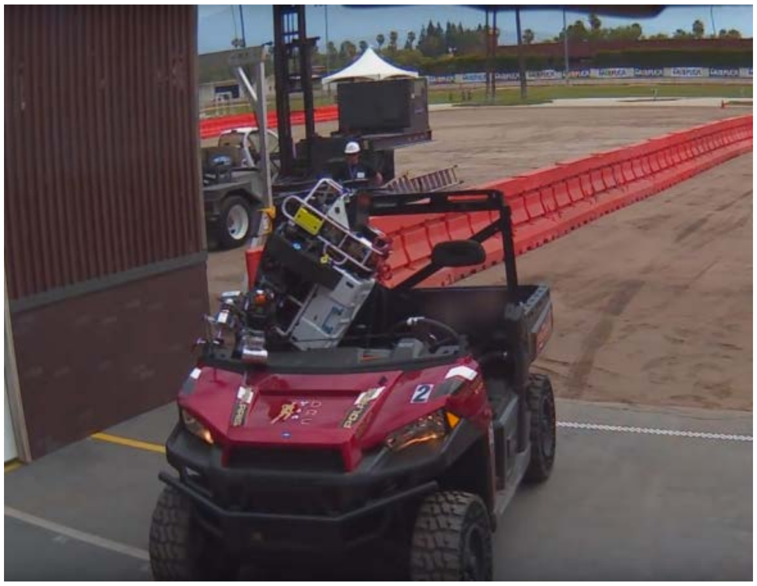


Figure 74. Atlas Reaches Driving Goal (Run 2)

4.2.2 Egress. On our first run, we initiated egress without issue. However, our field team noted that our Atlas was in an unsafe situation once the initial stage linear guide had extended. It is suspected that our Atlas was incorrectly seated on its mount initially and fell backwards off of the mount when the mechanism actuated. A reset was called, denying us of the task point. On the second run, egress worked flawlessly. As pictured in Figure 75, the slide lowered Atlas smoothly onto its feet. As our most operator-intensive task, our many hours of practice obviously showed their reward.

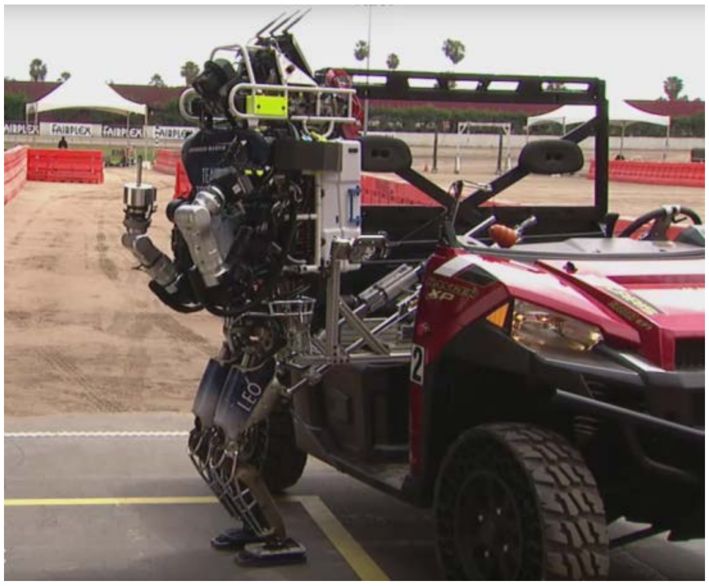


Figure 75. Atlas Egresses (Run 2)

4.2.3 Door. On both runs, we were able to place and walk to our door using the automated detection and walk to behaviors. On our first run, we found that the door handle for the course was difficult to turn (as had been noted by our walkthrough team). It took several attempts teleoperating the left hand against the handle to open the door. We walked through without issue and scored the task point. This successful opening and passing through the door is shown in Figure 76.

On the second run, we found that the left wrist was inoperable post vehicle egress. Because our operator had some practice opening the door with right hand POKEY stick, we were able to open the door through a combination of arm teleoperation and squatting commands. We had not recently practiced rising from the resulting extremely low squat height and our Atlas robot fell as we attempted to do so.

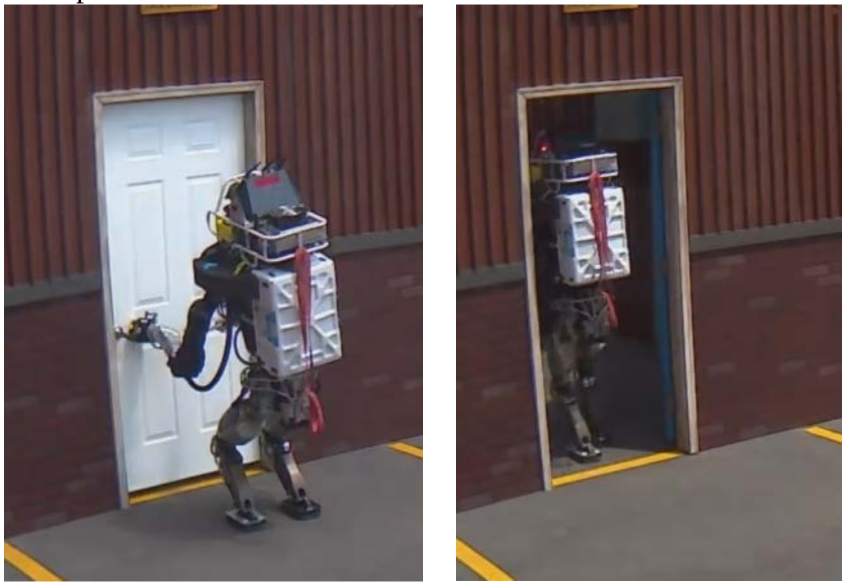


Figure 76. Atlas Opens and Clears Door (Run 1)

4.2.4 Valve. We only made it to the valve task on the first run. We placed our POKEY stick between the spokes of the valve. However, we found that we were unable to successfully send our valve-turning trajectory command over our communication link successfully. We had gotten the communications working only the night before and had not been able to practice with the competition communications restrictions at this point. As we were attempting to send the command, the trial came to an end, denying us the point. Figure 77 shows our position at the end of the run. We addressed the issue that night, planning the motion trajectories on both the UI and robot sides of the system, so as not to have to send the trajectory over the narrow bandwidth channel.

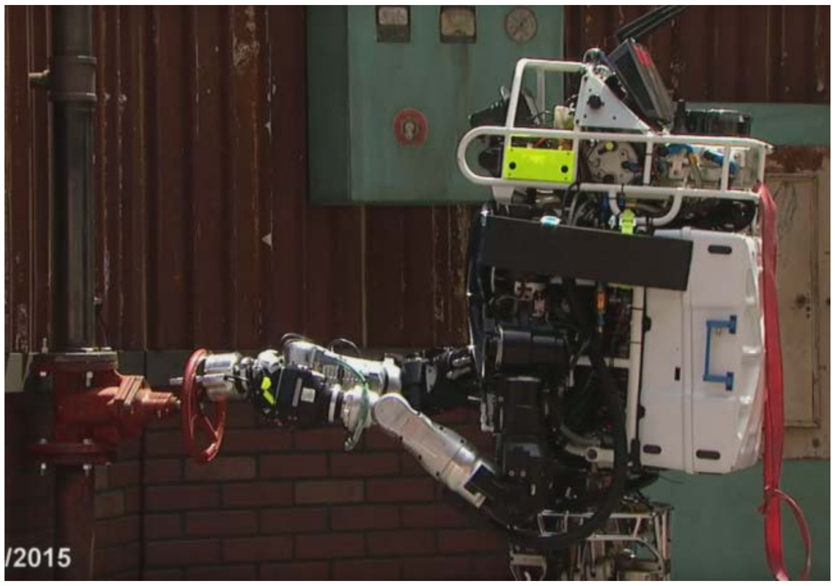


Figure 77. Atlas Ready to Turn Valve (Run 1)

5 CONCLUSIONS

We believe we took a different approach from most teams. Rather than starting with a purely teleoperated system and automating only the simple or repetitive tasks, we envisioned a system that would default to automated actions and only request operator input or teleoperation when it determined itself to be incapable of completing an objective. Instead of asking “What is the bare minimum we need to automate to accomplish the tasks in the time allotted?” we asked “How can we make the system as autonomous as possible while still guaranteeing task completion amidst uncertainty?”

To work effectively with the robotics community, we adopted the open source Robot Operating System (ROS). As a relatively small team, we heavily leveraged existing ROS components for the Virtual Robotics Challenge. This was very important for us in order to deliver a working system with the time constraints imposed by the competition. As we proceeded on with the DRC Trials and Finals, we decreased our reliance on stock ROS components and wrote our own. For example, we initially used the ROS SMACH state-machine framework before writing our own behavior manager. We initially used the ROS Move-It motion planning stack before creating our own lighter-weight version that incorporated additional features like Task Space Regions for grasp sampling. Additionally, we have wrapped existing ROS components in ways that add functionality, such as our adaptive perception manager, which provides easy methods to start, stop, and chain together sequences of perception algorithms. One lesson we learned is that UI development is very time-intensive. We would have saved time had we started with the ROS rviz visualization tool and modified it, rather than writing our own UI from scratch.

While we would have liked to have performed better in the DRC Finals, we have a good foundation for a system that utilizes the human as more of a supervisor than a teleoperator. We refer to this as human-guided autonomy. We have not yet utilized the notion of task execution confidence. Instead, when pressed, we reverted to teleoperation and did not give our autonomy framework a chance. This can also be attributed to lack of practice time. We needed more time and practice to determine reliable parameters for the knowledge base used by the reasoner. There is also potential here to use demonstration learning to teach the robot new tasks.

To be more useful, automation components such as ours need to be more generalizable. For a competition like the DRC Finals, it was tempting (and feasible) to over-fit the solution to the problem. We were able to encode many properties of the task setup before the robot ever entered the field. One avenue for future research is to better gather this information from vision and other sensor data to create more robust low level behaviors in mobility and manipulation.

In the competition, we ended up spending so much time on driving and egress (over half of our allotted time) that we were unable to test the rest of our system. We needed to spend more time and effort developing the driving components. Indeed, our biggest takeaway was that we needed more time to practice all the tasks. Not practicing the tasks under competition settings was very detrimental, as it would have given us time to address the bugs exposed. Unfortunately, our communications manager only became functional the day before the competition began.

Future research needs to focus on robust perception capabilities to provide monitors that can close the loop on autonomous behaviors. For example, automated vision to recognize that a door has been opened and vibro-tactile perception to feel that a cutting tool is activated are critical functionalities for the robot to become more autonomous. More advanced low-level humanoid controls incorporating visual feedback are necessary for platforms like Atlas to operate at higher speeds and in dynamic environments. Dexterous manipulation with multi-fingered grippers is required for robots to be able to perform more than a limited subset of manipulation tasks. Algorithms for in-hand object estimation and control incorporating tactile sensing are necessary, in addition to robust tactile sensors that last more than a few minutes. Finally, automated population of a knowledge base through observation of humans would speed up the process of adapting a robot to a new problem domain.

6 REFERENCES

Berenson, D., Srinivasa, S., & Kuffner, J. (2011). Task Space Regions: A Framework for Pose-Constrained Manipulation Planning. *International Journal of Robotics Research*, 1435-1460.

Cacace, J., Finzi, A., & Lippiello, V. (2014). A mixed-initiative control system for an Aerial Service Vehicle supported by force feedback. *Intelligent Robots and Systems (IROS),* (pp. 1230 - 1235).

Carey, M. W., Kurz, E. M., Matte, J. D., Perrault, T. D., & Padir, T. (2012). Novel EOD robot design with dexterous gripper and intuitive teleoperation. *World Automation Congress*, (pp. 1-6). Puerto Vallarta.

Chaomin, L., Yang, S. X., Krishnan, M., & Paulik, M. (2014). An effective vector-driven biologically-motivated neural network algorithm to real-time autonomous robot navigation. *Robotics and Automation (ICRA)*, (pp. 4094 - 4099).

Desai, J. P., Ostrowski, J., & Kumar, V. (1998). Controlling formations of multiple mobile robots. *Robotics and Automation (ICRA)*, (pp. 2864 - 2869).

Diankov, R. (2010). Automated Construction of Robotic Manipulation Programs. *PhD Thesis*. Robotics Institute, Carnegie Mellon University.

Geiger, A., Ziegler, J., & Stiller, C. (2011). Stereoscan: Dense 3D Reconstruction in Real-Time. *Intelligent Vehicles Symposium.*

Goodrich, M. A., & Schultz, A. C. (2007). Human-robot interaction: a survey. *Foundations and Trends in Human-Computer Interaction*, 203 - 275.

Henry, C. (2009). The meta state machine (MSM) library. *Boost Libraries Conference.* Aspen.

Lomas de Brun, M., Zaychik Moffitt, V., Franke, J. L., Yiantsios, D., Housten, T., Hughes, A., et al. (2008). Mixed-initiative adjustable autonomy for human/unmanned system teaming. *AUVSI Unmanned Systems North America.*

Murphy, R., Kravitz, J., Peligren, K., Milward, J., & Stanway, J. (2008). Preliminary report: Rescue robot at Crandall Canyon, Utah, mine disaster. *Robotics and Automation (ICRA)*, (pp. 2205-2206). Pasadena.

Ott, C., Roa, M., & Hirzinger, G. (2011). Posture and Balance Control for Biped Robots Based on Contact Force Optimization. *IEEE-RAS International Conference on Humanoid Robots*, 26-33.

Pan, J., Chitta, S., & Manocha, D. (2012). FCL: A General Purpose Library for Collision and Proximity Queries. *IEEE International Conference on Robotics and Automation*, (pp. 3859-3866).

Pomerlau, F., Colas, F., Siegwart, R., & Magnenat, S. (2011). Tracking a Depth Camera: Parameter Exploration for Fast ICP. *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (pp. 3824-3829).

Pratt, J. (2015). Team IHMC's Lessons Learned from the DARPA Robotics Challenge Trials. *Journal of Field Robotics*, 192-208.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach.* Pearson Education.

Rusu, R., & Cousins, S. (2011). 3D Is Here: Point Cloud Library (PCL). *IEEE International Conference on Robotics and Automation.*

Smits, R. (2015, June). *KDL: Kinematics and Dynamics Library*. Retrieved from http://www.orocos.org/kdl

Sucan, I., Moll, M., & Kavraki, L. (2012, December). The Open Motion Planning Library. *IEEE Robotics and Automation Magazine*, pp. 72-82.

Tedrake, R. (2014). *Drake: A Planning, Control, and Analysis Toolbox for Nonlinear Dynamical Systems*. Retrieved from http://drake.mit.edu.

Zhang, L., Lee, S.-L., Yang, G.-Z., & Mylonas, G. P. (2014). Semi-autonomous navigation for robot assisted tele-echography using generalized shape models and co-registered RGB-D cameras. *Intelligent Robots and Systems (IROS),* (pp. 3496-3502).

7 LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

Air Force Research Laboratory (AFRL)
DARPA Robotics Challenge (DRC)
DRC Simulator (DRCSim)
Emergency Stop (E-Stop)
Free Open Source Software (FOSS)
Institute for Human & Machine Cognition (IHMC)
Inverse Kinematics (IK)
Iterative Closest Point (ICP)
Kinematics and Dynamics Library's (KDL's)
Observe-Orient-Decide-Act (OODA)
Open Dynamic Engine (ODE)
Operator Control Unit (OCU)
Pointed Object for Kinematic Extension without Yielding (POKEY) Stick
Point Cloud Library (PCL)
Robot Control Unit (RCU)
Robot Operating System (ROS)
ROS State-Machine (ROS SMACH)
Simulation Construction Set (SCS)
Trusted Remote Operation of Proximate Emergency Robots (TROOPER)
User Interface (UI)
Virtual Robotics Challenge (VRC)
Worcester Polytechnic Institute (WPI)
Zero Moment Point (ZMP)