Humanoid Robot Navigation and Obstacle Avoidance in Unknown Environments

Griswald Brooks

June 10, 2013

Abstract

Real-time path planning is a necessity for any autonomous mobile robot operating in unknown environments. This report describes an implementation of the Game-Theoretic Optimal Deformable Zone with Inertia and Local Approach (GODZILA) path planning algorithm on the Nao humanoid robot by Aldebaran Robotics. The algorithm is a lightweight local path planner that does not require building an environment map. Forward facing ultrasonic distance sensors on the Nao were used for occlusion detection. Theory, simulation, and implementation results will be discussed.

Contents

List of Figures

Chapter 1

Introduction

Humanoid robots garner much interest not only in the fields of engineering and science, but over a wide range of social and non-technical contexts. [1][2] The humanoid form makes robots more relatable and accepted by the general public. Socially assistive and service robots need this sort of acceptance in order to perform their jobs more effectively. [3] They also offer unique opportunities for robots to operate in environments which are commonly designed for humans. These environments are typically difficult for many robots to operate in due to things such as narrow corridors, varying heights of door knobs and handles, and stairs. With the ability to traverse a wide range of terrains from rubble to stairs and the dexterity offered by multiple degree-of-freedom manipulators and fingers, humanoids as a platform have a flexibility not seen in many mobile robots. This makes them desirable for operating not only in human structured environments, but unstructured environments such as construction and disaster scenarios. [4]

With such mechanical flexibility come many challenges. Efficient and adaptable gaiting for humanoids is an active and heavily studied area of research [5][6], as well as the perception capabilities necessary to take advantage of such robust possible modes of operation. Facial, object, and speech recognition, localization, mapping, path planning, neural and semantic networks are just some of the perception areas researched in order to get humanoids to operate in natural human environments. [7][8][9][12] Aside from the algorithmic challenges of leveraging humanoid platform capabilities, designing platforms with efficient mechanical and electrical systems to accommodate difficult weight and power requirements make this field rife with interesting and novel research possibilities. Vision, structured light, planar laser scan-

ning, series-elastic actuation, and harmonic drive systems are a few of the sensing and actuation topics particularly applicable to humanoid research. [13][14]

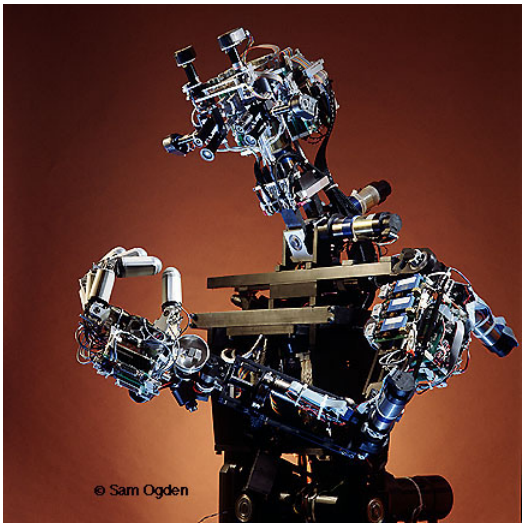

Figure 1.1: Cog Humanoid Robot at the MIT Artificial Intelligence Laboratory.

For this project, the Nao humanoid platform by Aldebaran Robotics is used [17]. Nao is approximately two feet tall, with 25 degrees-of-freedom, and various sensors including HD cameras, ultrasonic distance sensors, and inertial sensors. It is easily programmed with a graphical programming interface called Choreographe, or through a framework called NAOqi in which multiple text-based languages such as C++ and Python can be used. Through either API, prebuilt routines such as gaiting and object tracking can be used, allowing for other areas utilizing such commonly needed tasks to be explored, though custom algorithms can be written to supplement or replace the prebuilt ones.

In this project, the topic of path planning and obstacle avoidance is explored. The path planning problem consists of moving a mobile platform or end-effector from a start location to a goal location. Examples vary from ground robots moving through an office building, UAVs navigating to GPS waypoints, or end-effectors avoiding objects on a cluttered table to grasp an object [10]. Methods of solving such a problem depend largely on the construction and knowledge of the environment and the knowledge the robot

has about its position within the environment. For example, a mobile robot operating in the plane within a static environment, *a priori* map, and direct position sensing has different challenges than a mobile robot operating in three-dimensions, within an unknown dynamic environment, and only platform velocity estimation. Popular approaches include graph search algorithms such as A* or D*, Bug algorithms, and potential fields approaches [11]. Potential fields algorithms treat the mobile robot as a point mass that is repelled by obstacles and attracted by the goal.

The algorithm implemented is the Game-Theoretic Optimal Deformable Zone with Inertia and Local Approach (GODZILA) algorithm [16]. It is a potential fields style algorithm, with additions that add a form of inertia to reduce limit-cycles and local mimima, a straight-line planner for when the goal is insight, and a goal-randomizer for when the robot is stuck. The ultrasonic sensors on the Nao were used to estimate distance to obstacles, while the prebuilt red ball tracker routine that uses Nao's HD cameras to estimate range and bearing to a 6 cm diameter red ball was used to indicate the location of the goal. Prebuilt gaiting algorithms were used to command the Nao to linear and angular velocities. This allowed the Nao to walk towards a goal with multiple obstacles interfering in its path.

Results from this work were published in [15].

This report is organized as follows: Chapter 2 discusses the GODZILA path planning algorithm, Chapter 3 reviews applicable topics of the Nao platform, Chapter 4 presents simulation methodology and results, Chapter 5 presents results from implementing the path planning algorithm on the Nao, and Chapter 6 states conclusions and closing remarks.

Chapter 2

GODZILA Path Planning Algorithm

GODZILA [16] is a general computationally lightweight path planning and obstacle avoidance algorithm that does not require any a priori knowledge of environment and does not rely on building of an obstacle map.

GODZILA is based on three components:

- Optimization based on a cost that penalizes motion in directions other than the direction to the target, motion towards obstacles, and back-tracking.
- A local straight-line planner utilized if the target is visible.
- Navigation towards a random target when a local "trap" is detected.

The desired motion direction is generated through online minimization of an optimization cost at each sampling instant. It can be shown that the optimization cost can be chosen so that the minimizer can be obtained in closed form. The optimization cost has three terms which penalize, respectively, motion in directions other than the direction to the target, motion towards obstacles, and back-tracking. In addition to the optimization algorithm, GODZILA includes two components, a local straight-line planner utilized if the target is visible and navigation towards a random target (utilized when local minima or traps are detected).

GODZILA can utilize obstacle range measurement data measured using a variety of sensors such as ultrasonic sensors or laser range finders or obstacle

ranges estimated using sensors such as camera, Radar, etc. The GODZILA algorithm is formulated in terms of a set of directions in which obstacle range measurements are measured/estimated. Denoting by q_i, the unit vector defining the i^{th} sensor direction, the obstacle range measurements/estimates are denoted as $s_i, i = 1, \ldots, n$; $s = (s_1, s_2, \ldots, s_n)$. x_p and x_h are used to denote the current position and the unit vector along the current heading, respectively, and u_e is used to denote the obstacle set (possibly time-varying) of the environment, i.e., the set of points in the environment occupied by obstacles. The sensors in the GODZILA algorithm are defined as returning the distance from the current position to the obstacle set in the directions of the sensors. The sensor measurements can be modified to enforce a clearance zone by exaggerating the physical dimensions of the robot. Since this exaggeration is only relevant when the obstacles are close to the robot, introducing, for instance, $s'_i = s_i - p_i e^{-as_i}$ with $p_i, i = 1, \ldots, n$, and a being positive constants, we have $s'_i \approx s_i$ in the far zone and $s'_i \approx s_i - p_i$ in the near zone. This weighting induces large penalties when obstacles are near and small penalties when obstacles are distant.

The path planning objective is essentially to generate a trajectory that tracks a target trajectory $x_d(t)$ (nominally a goal location x_d) while avoiding the obstacle set $u_e(t)$. The GODZILA path planning and obstacle avoidance algorithm [16] achieves this objective using a combination of an optimization procedure, a random walk, and approach to a visible target. At any time, the optimal heading is computed by an optimization procedure using as effective target one of the following

1. The actual target x_d
2. A fictitious target location on an intermediate straight-line trajectory to the target x_d
3. A random target.

The optimization to find a desired motion direction is performed with respect to an objective function given by $J(y) = J_1(y) + J_2(y) + J_3(y)$ which has the following three components:

1. $J_1(y)$ - a component that penalizes motion in directions other than the target direction,
2. $J_2(y)$ - a component that penalizes motion towards obstacles, and

3. $J_3(y)$ - a component that penalizes back-tracking.

The first component $J_1(y)$ attempts to make the robot move in the direction of the target and is designed such that it is an increasing function in terms of $\left|\left|y - \frac{(x_d - x_p)}{||x_d - x_p||}\right|\right|$ and is a decreasing function in terms of $||x_d - x_p||$ and such that in the immediate vicinity of the target, $J_1(y)$ is dominant in the objective function $J(y)$. The second component $J_2(y)$ attempts to prevent motion in directions that would bring the robot to the proximity of obstacles. Direction-weighted terms [16] are usually included in $J_2(y)$ to model the property that the effect of this component is more important in directions that are either along the current heading or along the straight-line heading to the target. This component is designed to also encourage motion in directions along which obstacles are far away, especially if such a direction is along the straight-line heading to the target, and also such that in the close vicinity of obstacles, the effect $J_2(y)$ of obstacles predominates in the objective function. The third component $J_3(y)$ is designed to be an increasing function in terms of $||y - x_h||$, thus penalizing changes in heading and is inspired by the physical notion of inertia. $J_3(y)$ is instrumental in preventing limit-cycle oscillations (wherein, without $J_3(y)$, oscillatory motions can arise with the obstacle avoidance incentive and the target-reaching incentive alternatively becoming dominant). $J_3(y)$ essentially makes a potential limit cycle spatially larger and hence provides better chance of finding a way around a blocking obstacle. If the objective function $J(y)$ has a unique minimum over the unit ball $\mathcal{B} = \{y : ||y|| = 1\}$, the output of the algorithm is the unique minimizer. If the minimizer is not unique, the output is defined as one which gives the smallest $J_3(y)$.

It was shown in [16] that the minimizer (i.e., the optimal heading) of the objective function $J(y)$ can be found in closed form if quadratic forms of functions are utilized within components appearing in $J_1(y)$, $J_2(y)$, and $J_3(y)$, in which case the closed form optimal solution can also be equivalently interpreted in terms of force components towards the goal, away from obstacles, and in the current motion direction (i.e., the inertia component). The desired heading found from the optimization algorithm is translated into a heading rate that takes into account the kinematic capabilities of the robot. Also, the linear velocity of the robot is designed to be such that it is small in the close vicinity of obstacles and in the proximity of the target location.

As mentioned before, GODZILA also incorporates two additional components in addition to the optimization-based computation of the optimal

heading, i.e., a local straight-line planner utilized if the target is visible and navigation towards a random target. At any time, if the target is visible, i.e., the nearest obstacle in the direction of the target is estimated to be farther than the target, then the robot can proceed in a straight-line towards the target. However, since a direct straight-line path to the target may bring the robot in close proximity with obstacles, the straight-line path to the target is instead prescribed as a desired trajectory and the navigation along this path is performed using the optimization-based approach described above. This local straight-line based virtual reference trajectory is of particular use when the robot needs to pass through a narrow corridor to approach the target. The random target component addresses possible limit cycle oscillations. While the introduction of the inertial term in the optimization cost has the effect of increasing the spatial extent of possible limit cycles, limit cycle oscillations can still occur if the obstacle set is spatially large since the GODZILA algorithm is based on local sensor data and no map of the environment is built. A local limit cycle or a *trap* can be detected using, for instance, the variance of the position variable over some number of successive sampling instants. Alternatively, the distance to the target can be used as a metric and a trap situation can be inferred if this distance does not change significantly over some period of time. If a trap is detected at a time t_0, a random target location $\hat{x}_d$ is chosen and the navigation is performed for a time duration T_{roam} using the optimization procedure described above with x_d replaced by $\hat{x}_d$. If a trap situation is detected multiple times, then progressively longer sequences of random moves are utilized to escape the local minima.

2.1 Virtual Sensors

In the application to the humanoid robot navigation problem using ultrasonic sensors considered here, a primary sensory challenge is, as described earlier, the wide beam angle which results in large angular uncertainty as to where the sonar returns are from (i.e., the X-Y locations of the points in the environment that correspond to a single measured distance from each of the ultrasonic sensors). To address this angular uncertainty, the obstacle range measurements/estimates for the GODZILA algorithm are generated as a combination of actual and virtual sensor measurements as follows:

1. For each ultrasonic sensor with a beam width of θ_w (which is 60^o for

the ultrasonic sensors on the NAO robot), if the corresponding distance measurement at a sampling time is d, this distance measurement is mapped to multiple sensing directions spanning the beam width (typically, 5 directions, i.e., angular offsets of 0, $\pm k\theta_w/2$, $k = 1, 2$, relative to the center of the sensor beam). The distance corresponding to the center of the sensor field of view (i.e., offset of 0 degrees) is defined to be d and the distances corresponding to the other *virtual* sensor directions are defined to be larger, e.g., $(1 + ak)d$, $k = 1, 2$, with a being a positive constant to model the decreasing confidence that the returned distance values are due to obstacles at the edges of the beam.

2. The sensor measurements from a few prior sampling instants are also utilized as virtual sensor measurements for the current time instant. The robot-relative obstacle locations corresponding to the multiple virtual sensor directions as described above that are modeled for each ultrasonic sensor are utilized along with estimates of the robot pose (position and heading) at the previous time steps. These obstacle locations are mapped into the current robot-relative frame at the current time instant by utilizing estimates of relative translations and rotations over successive sampling instants. The directions corresponding to these obstacle locations are also now utilized as virtual sensor directions with the corresponding estimated ranges for the GODZILA algorithm; hence, the range measurements for use in the GODZILA algorithm is comprised of measurements over a sliding window of time including the current and a few past sampling instants. Denoting the time horizon utilized by n_h and using 5 virtual sensor directions per sonar to model the beam width as described above, the total number of virtual range sensors (taking into account both the ultrasonic sensors) utilized in the GODZILA algorithm is $2(5)n_h = 10n_h$. To model the reduction in confidence on the measurements from prior sampling times projected into the current robot pose, the weights for sensor directions corresponding to older time instants can be defined to be smaller in the optimization component $J_2(y)$ utilized in the GODZILA algorithm to penalize motion towards obstacles.

More general variants of these concepts of virtual sensors are currently being addressed in ongoing work including the incorporation of additional weighting terms on virtual sensor directions that are more often perceived as

having closer obstacles. While implementation of the virtual sensors as described above requires keeping a memory of a sliding window of time of sensor measurements and relative robot poses, the resulting computational requirements (both memory and processing requirements) are significantly lower than for incrementally building an environment map during robot operation. Thus, the approach proposed here provides a computationally light-weight technique that can be easily implemented on small embedded platforms.

Chapter 3

Platform

The platform utilized on this project is the Nao humanoid experimental robotic platform by Aldebaran Robotics. It is a 25 degree-of-freedom humanoid mobile manipulator with two HD cameras, two ultrasonic distance sensors, four microphones, one three-axis accelerometer, two one-axis gyroscopes, and four force sensors in each foot. The platform is programmed using a framework called NAOqi allowing development using various languages including C++ and Python.

3.1 Distance Sensors

Two ultrasonic sensors in the chest allow for distance measurements to occlusions. Each sensor returns a single reading per measurement update as a distance reading in meters. This measurement is intended to represent the distance to the closest object within the sonar's field of view. The transmitters are mounted at an angle of 20 degrees from the forward direction of the Nao and the receivers are mounted at 25 degrees. They have a 60 degree viewing angle with a detection range between 0.25 and 2.55 meters with a 1 cm resolution.

As shown in Figure 3.2, the regions covered by the two ultrasonic sensors overlap. As the sensors do not report the angular position of the occlusion within the cone, uncertainty about the location of detected objects is high as they could be anywhere within the sonar cone.

The distance measurements also suffer from multipath returns and specular reflections which cause the robot to read incorrect distances at certain

Figure 3.1: Nao robot at Control/Robotics Research Lab

angles. For example, at times when the Nao is oriented toward a wall at an obtuse angle, the right distance sensor, which was closer to the wall, will read a distance much larger than that of the left sensor, causing the robot to gravitate into the wall. An example of the problem can be seen in Figure 3.3.

At times, the source of erroneous sensor measurements stem from the material that the occlusion is made from. It was found that if the material is too soft, the distance reported to it will be larger, or at times, not be seen. Harder materials seem to correct for this. Soft materials are items like plastic trash cans. Metal trash cans respond more reliably. A brief overview about various issues with sonar can be read about in [23].

3.2 Programming

There are three ways to program the Nao. The first is to use the graphical environment Choreographe. It is a block-based programming method where the user drags modules into an area and connects the modules together with

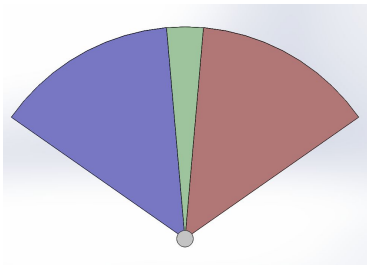

Figure 3.2: Nao sonar cones. The left cone is shown in blue and the right cone is shown in red. The green cone shows the overlap between the two.

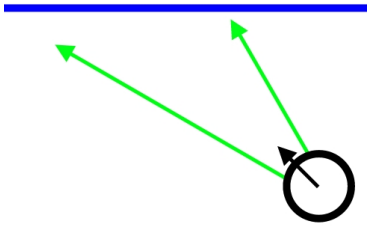

Figure 3.3: Distance measurement error at obtuse angles. In the figure, the black circle represents the Nao in the plane, while black arrow represents the current heading. The blue line represents a wall and the green arrows indicate the direction the sonars are pointing. When in positions similar to this the right sensor would report a distance greater than the left sensor, even though the right sensor is closer to the wall.

arrows, to create a program flow. These modules can either be prebuilt or made by the programmer. The programmer can make their own modules by either dragging basic modules together and conglomerating them into a new module or they can program these modules using a text-based language, such as Python. The second and third ways each involve using the framework provided by Aldebaran Robotics called NAOqi. NAOqi allows for many functions necessary for roboticists such as parallelism, resource management, and event synchronization while being cross-platform and cross-language[21]. This API allows the programmer to take advantage of more of the resources available on Nao. NAOqi can be utilized in two ways. Either the projects built with NAOqi can be compiled for the linux distribution on Nao and then uploaded to it and executed, or the project can be compiled and run on an

external computer and the commands sent to Nao over a network connection. For this work, code is not compiled on the Nao itself but rather on an external computer and commands are sent to the Nao via a Wi-Fi network that both the Nao and the external computer are connected to. After setting up the build system qiBuild [22] on the external computer, a new project is built using qiBuild and then edited and complied using Visual Studio 2008. Once built, the executable from the project is run from the command line with the IP address of the Nao as an argument and the Nao responds. Any print statements are seen on the command line.

The NAOqi framework uses the concept of proxies in order to access sensor data or send movement commands. Proxies are objects where data is accessed, stored, or sent. They are proxy to where these items are actually used or accessed, which are the Nao's memory, referenced using memory keys [24]. For example, the memory keys for accessing the distance measurements for the sonar are:

```
1  Device/SubDeviceList/US/Right/Sensor/Value
   Device/SubDeviceList/US/Left/Sensor/Value
```

More about the sonars can be read about in [25].

To initiate the updating of the sonars, the sonar proxy must be initialized with the IP address of the Nao and then subscribed to by the module. The module parameter is simply a keyword for your application. It is of your choosing. In this case the string "RAPlanner" was chosen. This will allow us to poll the distance readings rather than having to subscribe to events and provide callback functions.[26][27]

```
   AL::ALSonarProxy sonarPrx(argv[1], 9559);
2  sonarPrx.subscribe("RAPlanner");
```

9559 is the default port that NAOqi listens to. argv[1] contains the IP address of the Nao, as it is the first parameter in the command line argument when the executable is called. To access the sensor data a memory proxy must be used as this data is stored in memory.

```
   AL::ALMemoryProxy memPrxSonar(argv[1], 9559);
```

In the main loop, these values are accessed through methods of the proxy.

```
1  rightDist = memPrxSonar.getData(keyR);
   leftDist = memPrxSonar.getData(keyL);
```

where keyR and keyL are the const string key values of where the data is stored, "Device/SubDeviceList/US/Right/Sensor/Value" and "Device/SubDeviceList/US/Left/Sensor/Value" in this case. rightDist and leftDist are of type AL::ALValue which are later cast to floats for use by the path planning algorithm. The data is returned in meters.

To send motion commands to the Nao, a motion proxy must be used. The proxy must be initialized and the walk can be started.

```
   AL::ALMotionProxy motionPrx(argv[1], 9559);
2  motionPrx.walkInit();
```

As the Nao has not been given any movement commands, it will not move. Movements can be sent to it using a number of APIs [19]. The one used for this project was:

```
motionPrx.setWalkTargetVelocity(Vx, Vy, Om, stepFreq);
```

The method takes three different velocity commands, forward, lateral, and angular, in terms of fraction of maximum step length, and step frequency. Vx, Vy, Om, stepFreq were floats passed into the method, updated by the path planning algorithm on every iteration of the main loop. The maximum step length is 8 cm forward, 16 cm laterally, and 0.523 radians angularly. The maximum step frequency is 2.381 Hz. Due to a stability controller in the built-in gaiting algorithm for the Nao, it takes 0.8 seconds for the robot to react to new commands. At maximum step frequency, this equates to about 2 steps [18]. The Nao can be stopped by setting all of the target velocities to zero.

For goal estimation, the inbuilt red ball tracker API is used. The red ball tracker returns the estimated position of a 6 cm red ball in 3-space, in the frame of the Nao's torso in meters [29]. In addition to the red ball tracker proxy being initialized and the tracker started, the Nao's head stiffness needs to be set. "Stiffness" refers to the percentage of available torque used in order

to have a joint reach a target angle [28]. The "Head" is a kinematic chain which refers to a collection of joints, in this case the yaw and pitch axes of the head [20]. A stiffness of 1.0 instructs the Nao to use all available torque to control the joint.

```
motionPrx.setStiffnesses("Head", 1.0f);
AL::ALRedBallTrackerProxy redBallPrx(argv[1], 9559);
redBallPrx.startTracker();
```

Following this, in the main loop new data is checked for and then accessed.

```
if(redBallPrx.isNewData()){}
  ballPose = redBallPrx.getPosition();
}
```

getPosition() returns a vector of floats in x,y,z which was then passed to the path planning algorithm as the goal location.

A number of other program enhancements were made including initial head orienting and being able to push Nao's head buttons in order to set velocities to zero. All of these enhancements had a similar structure to the above code and can be read in Appendix B

Chapter 4

Simulation

Simulating the robot in sample environments allows for more rapid algorithmic development and excludes implementation details which may masquerade as errors in the algorithm. The simulation was written in MATLAB for this project. The simulation constructed a virtual environment using line segments and approximated the robot as a point on the plane. A simple kinematic model of the robot is used, as modeling the full kinematics of the Nao robot is not necessary for the efficacy of the algorithm. A basic approximation for inertia is made by low pass filtering all velocity commands. Motion noise is injected by adding uniform random noise to the linear and angular velocities. The resultant velocities are integrated to gain robot position using a first order approximation. The sonar beams are approximated as cones in the plane with the sensor model returning the distance to the closest object within the region of the cone. These ranges are also perturbed by uniform random noise before being returned to the navigation algorithm. The goal location is a randomly generated point in the map whose location the robot was aware of at all times.

4.1 Kinematics

As the motion API for the Nao allowed commanding forward and angular velocities, the simple model used assumed that at each time step integration of these velocities would move the robot along the plane. Uniform random noise is added to these velocities to account for errors in encoding and motion. Dampening is added to simulate inertia in the robot's motion as a

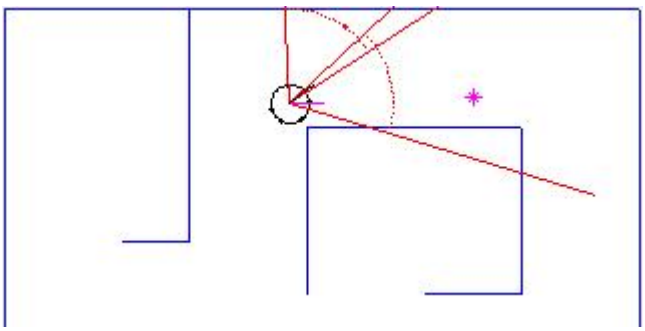

Figure 4.1: Planar robot simulation example. The robot is indicated by a black circle with the red cones simulating ultrasonic sensor beams and their detected range indicated by dotted red arcs in the red cone. Blue lines represent obstacles. The magenta line points towards the goal location, which is indicated by a magenta star.

simple low pass filter.

```
%%% MOTION NOISE %%%
v = v + .2*rand(1);
om = om + .01*rand(1);

%%% MOTION DAMPENING %%%
v = 0.1*v + 0.9*vp;
om = 0.1*om + 0.9*omp;

%%% MOTION MODEL %%%
r_pose(3) = r_pose(3) + om*dt;
r_pose(1) = r_pose(1) + v*cos(r_pose(3));
r_pose(2) = r_pose(2) + v*sin(r_pose(3));
```

v, om, vp, omp are the current linear and angular velocities commanded by the path planning algorithm and the previous linear and angular velocities passed to the motion model, respectively. r_pose is a vector of the current x, y, θ of the robot in the plane. rand(1) generates a uniform random scalar between zero and one.

4.2 Sensor Model

The sensor model assumes that a single distance measurement should be returned which represents the range to the closest point on an occlusion in the sensor's field of view. It uses parameterized line segments transformed into the robot's frame as obstacles. Using closed form equations, it iterates through every object in the map and checks for several conditions on each object to find the closest point on that object within the field of view of the sonar and then returns the distance to the closest object in that sonar cone.

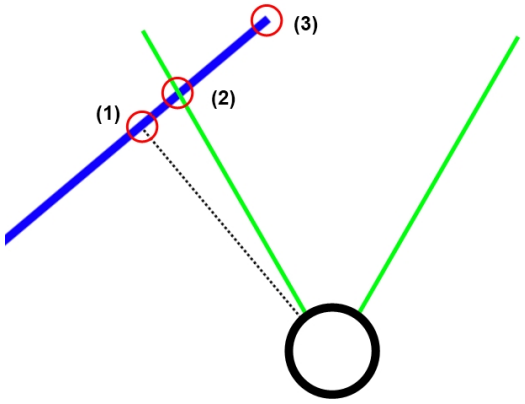

Figure 4.2: Sonar model points. The robot is indicated by a black circle with the green lines representing the field of view of the sonar beam. The blue line represents the line segment object and the red circles indicate the three points the sonar model calculates. 1) is the perpendicular point on the occlusion, 2) the intersection between the line segment and the beam edge, 3) the end points of the line segment.

Using infinite projections of the line segments, the points calculated are:

1. The perpendicular point on the line to the robot.
2. The beam edge intersection with the line.

These points were checked to be within the sonar cone and on the segmented line. Lastly, the if one of the segment's endpoints was in the sonar beam it too was considered. These points were all compared and the closest one was added to the range candidates for that beam. Once all the segments in the map were checked, the closest candidate was returned. Details of this model can be seen in the function getRangeBeam in Appendix A.9.

4.3 GODZILA Path Planner

Using the closed form solution for the cost functions, various force functions are used for the occlusions, goal, and inertia. Occlusion force magnitudes are generated according to an inverse square and then applied to the direction vector along the orientation of the respective sensor in the robot's frame.

```
f_range1 = -1/(RANGE_COEFF*((ranges(1) - RANGE_OFF)^2))*[cos(-
  base_angle),sin(-base_angle)]';
f_range2 = -1/(RANGE_COEFF*((ranges(2) - RANGE_OFF)^2))*[cos(
  base_angle),sin(base_angle)]';
f_ranges = f_range1 + f_range2;
```

RANGE_COEFF and RANGE_OFF were experimentally determined constants used to prescribe the rate at which the robot is repelled from occlusions and a distance for which the robot is asymptotically repelled from. base_angle and ranges are the orientation of the ultrasonic sensor and a vector of the returned ranges from the sensors. Despite the actually orientation of the sensors with respect to the robot (25 degrees), the base angle of the sensors in simulation and on the robot used was 45 degrees. When the actual angle was used, the robot had a difficult time navigating between obstacles. When this was changed to 45 degrees, both the simulation and the actual robot are able to traverse more narrow apertures more easily.

The goal force is calculated using a similar inverse square law. The range is calculated, as the goal location is generated in the world frame, and then applied to the inverse square as a min between the range and some maximum value, so that the influence of the goal never gets too small. This is then brought into the robot frame.

```
goal_range = sqrt((r_pose(1) - goal(1))^2 + (r_pose(2) - goal
  (2))^2);
f_goal = (1/((0.5*min(goal_range,25000))^2))*[goal(1) - r_pose
  (1), goal(2) - r_pose(2)]';
f_goal = [cos(-r_pose(3)),-sin(-r_pose(3));sin(-r_pose(3)),cos
  (-r_pose(3))]*f_goal;
```

As the inertia force was always in the direction of the current heading, it takes the form of a constant.

```
f_heading = 0.01*[cos(0),sin(0)]';
```

The forces are then summed and the resultant orientation vector converted into an angular rate.

```
f = f_ranges + f_goal + f_heading;
th_in = (atan2(f(2),f(1)));
om = TURN_MAGNITUDE*th_in*dt;
```

TURN_MAGNITUDE was an experimentally determined constant and dt is the time step used in the simulation.

The forward velocity is calculated in such a way that the robot will slow down when close to occlusions and speed up when far away. the *log* function is used for this so the robot will not go too fast when obstacles are very far away. It also causea the robot to push exponentially harder away from walls if it gets too close. The *exp* component slows the forward velocity when the robot has a large angular velocity. Finally, it is low passed with the previous forward velocity vp.

```
v_desired = 0.75*log(min(ranges)/30)*(5*exp(-om));
v = 0.1*vp + 0.9*v_desired;
```

A crude randomizer for trap detection was implemented using the variance of the naively integrated pose of the robot.

4.4 Results

Figure 4.3 show some results of the simulation in various environments using randomized start and goal locations. This figure shows the algorithm was effective at driving the robot to the goal.

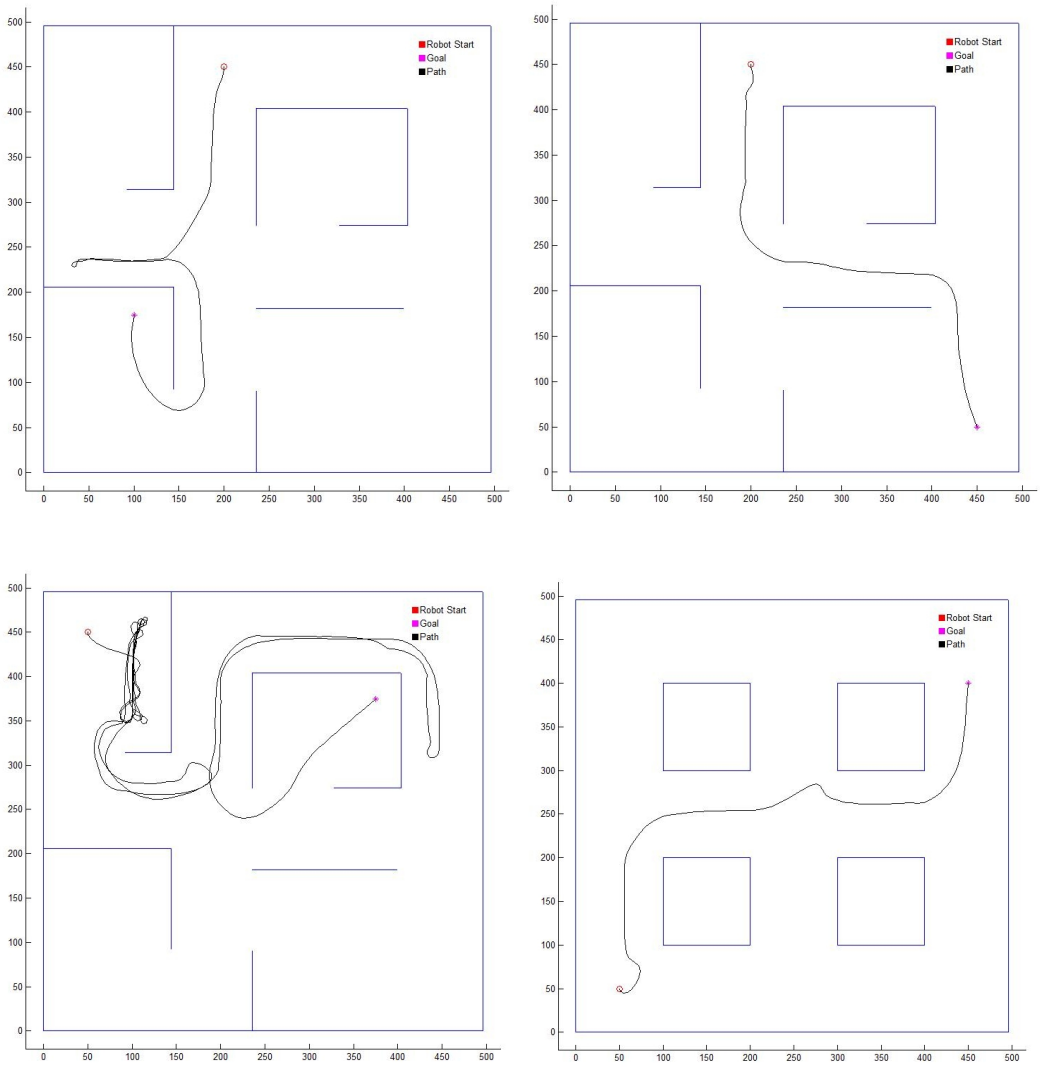

Figure 4.3: Simulation results. In these figures, the robot start and goal locations are shown in red and magenta, respectively. The trajectory of the robot is shown as a black line. The obstacles in the environment are shown as blue lines. The X and Y axis dimensions in the figures are in cm. Due to the large angular uncertainty inherent in the ultrasonic sensor measurements, some amount of meandering can be sometimes seen (as in the figure on the bottom left); however, the detection of local minima (or traps) and subsequent random walk components built into GODZILA ensure that the robot eventually finds a path to the goal location.

Chapter 5

Experimental Results

As presented in Chapter 3.1, ultrasonic distance sensors can, at times, present challenging issues. Multipath and specular returns were encountered and lead to data like that shown in Figure 5.1. This data was taken while Nao walked down a straight hallway, with no other obstacles to occlude its path. The inconsistency of the data, at times, interferred with the Nao's ability to navigate it's environment. Despite these inconsistencies, the Nao was able to path plan various sorts of office-like environments.

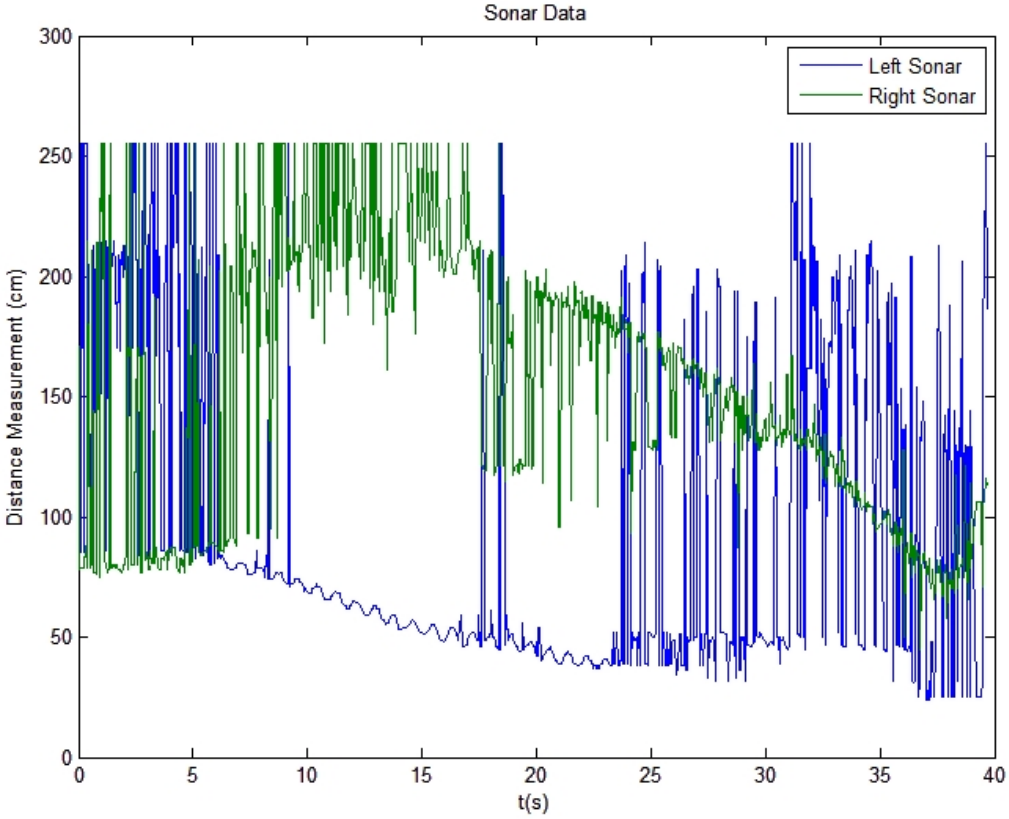

Figure 5.1: Nao sonar data. The plot shows the distance data reported from the ultrasonic sensors while the Nao walked down a corridor.

Figure 5.2 shows an experimental run of the Nao utilizing the GODZILA path planning algorithm. Nao was placed in a hallway setting with two cylindrical obstacles. The obstacles obstruct Nao's path to the goal, which was represented by a red cube measuring 10 cm x 10 cm x 8 cm. Nao is placed facing the goal. On start, Nao acquires the goal and walks towards it. Nao then guides around the first obstacle which sends it towards the second obstacle. Nao detects the second obstacle, guides around it, and walks to the goal.

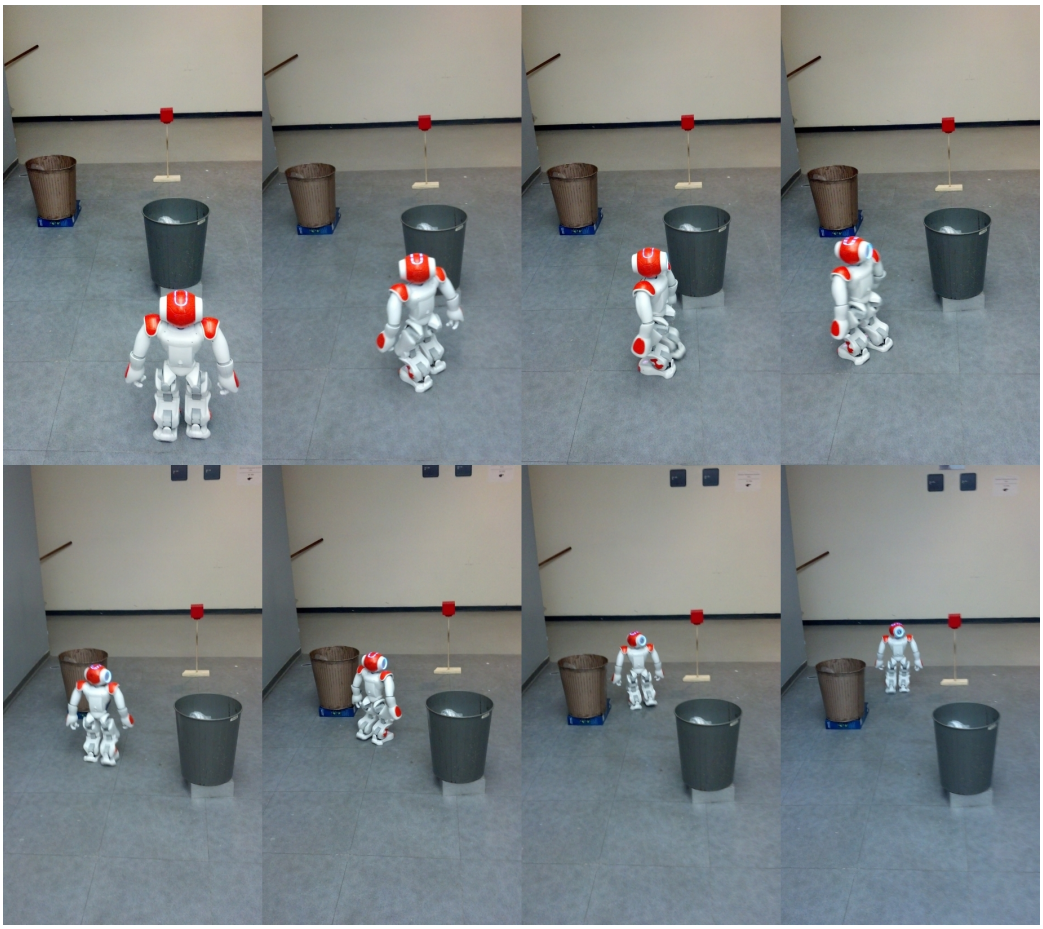

Figure 5.2: Nao time sequence. The Nao is started with the GODZILA path planning algorithm initiated. The goal location is indicated by the red cube. Two obstacles are obstructing the path to the goal. Nao sucessfully navigates around the obstacles to the goal.

Chapter 6

Conclusion

The GODZILA path planning algorithm has proven its effectiveness in allowing the Nao humanoid platform to navigate in office-like environments. While the Nao was able to sucessfully navigate, the vague and spurious nature of the distance data limited the effectiveness of the algorithm. Future work will add infrared distance sensors or some form of 3D sensor to the Nao to improve the quality and quanity of the range data for use with the algorithm. Adding magnetometers to combine with the onboard gyroscopes will allow for the creation of a better yaw estimate for the Nao. With this, a basic form of localized map can be implemented and GODZILA can use the map data to generate more comprehensive force vectors. The posibility for improved pose estimation will also allow for pose variance computations to be improved for the implementation of the trap detector. Additionally, a provision for goal location estimation for when the goal is not in sight or when sight of the goal has been lost will be explored.

Implementation of all of these features while running the software onboard the Nao will be researched in order to free the Nao from using an external computer and improve its range. Running the code onboard will also improve the speed of computation, which as the algorithms intensify, will be appreciable from running the code externally.

The Nao humanoid platform is a robust, multi-featured robot, which will allow for many types of algorithms to be tested, including gaiting and vison, as well as other types of path planners.

Appendix A

Simulation Code

A.1 Simulator

```
% This simulator models a differential drive robot moving in a planar
% environment. It displays the robot with some diameter, but the model does
% not take that into account, though there is some basic motion dampening
% modeled. The sensors are modeled as conical beams in the plane, returning
% the distance to the closest object.

cla, clc, clear

figure(1)
hold all
max_dim = 248; % Maximum field dimensions in centimeters
axis([-20,2*max_dim+20,-20,2*max_dim+20])
axis square

%%% FIELD WALLS %%%
field_walls = generateFieldWalls(max_dim);

%%% GOAL POSITION %%%
goal = 450*rand(2,1);
rand_goal = [0,0]';

%%% ROBOT VARIABLES %%%
HEADING_LENGTH = 25;
```

```
ROBOT_DIAMETER = 30;
r_pose = [100,225,(-90)*(pi/180)]'; % Starting pose of the robot
    [x,y,theta]
%r_pose = [375,375,(-90)*(pi/180)]'; % Starting pose of the
    robot [x,y,theta]
r_pose = [450*rand(1),450*rand(1),r_pose(3)];
v = 0;  % Linear Velocity, cm/sec
om = 0; % Angular Velocity, rad/sec

%%% BEAM VARIABLES %%%
VIEW_ANGLE = (60)*(pi/180);
MAX_BEAM_RANGE = 250; % 0.80 meters
MIN_BEAM_RANGE = 25; % 0.10 meters
% Beam Angles
base_angle = 45;
ang1 = -base_angle*(pi/180);
ang2 =  base_angle*(pi/180);
% Beam Angle Transformation Matricies
T1 = [cos(ang1),-sin(ang1), 0;
      sin(ang1), cos(ang1), 0;
               0,         0, 1];
T2 = [cos(ang2),-sin(ang2), 0;
      sin(ang2), cos(ang2), 0;
               0,         0, 1];

%%% ESTIMATED POSES %%%
poses = zeros(3,100);

%%% TIMER FOR STUCK DETECTION %%%
stuck_timer = StuckTimer();

%%% TIME VARIABLES %%%
dt = 0.25; % Time step
Tf = 5000; % Final Time

pause(1);

% *** ITERATE THROUGH POSITIONS *** %
for t = 0:dt:Tf
    cla

    %%% PLOT ROBOT %%%
    % Robot Transformation Matrix
    T = [cos(r_pose(3)),-sin(r_pose(3)), r_pose(1);
         sin(r_pose(3)), cos(r_pose(3)), r_pose(2);
```

```
                        0,                  0,          1];
68      plotRobot(T, ROBOT_DIAMETER, HEADING_LENGTH, 'k');

70      %%% PLOT WALLS %%%
        for iter = 1:2:length(field_walls)
72          line(field_walls(iter:iter+1,1),field_walls(iter:iter
        +1,2))
        end
74
        %%% PLOT GOAL %%%
76      plot(goal(1), goal(2), 'm*');

78      %%% CALCULATE RANGES %%%
        % Get ranges to walls
80      [range1, feature1] = getRangeBeam(field_walls', VIEW_ANGLE,
       MAX_BEAM_RANGE, MIN_BEAM_RANGE, T*T1);
        [range2, feature2] = getRangeBeam(field_walls', VIEW_ANGLE,
       MAX_BEAM_RANGE, MIN_BEAM_RANGE, T*T2);
82
        % Add sensor noise
84      range1 = range1 + 0.2*rand(1);
        range2 = range2 + 0.2*rand(1);
86
        %%% DRAW BEAM %%%
88      drawBeam(VIEW_ANGLE, MAX_BEAM_RANGE, range1, T*T1, 'r');
        drawBeam(VIEW_ANGLE, MAX_BEAM_RANGE, range2, T*T2, 'r');
90
        ranges = [range1,range2];
92
        % *** Robot Navigation *** %
94    vp = v;
        omp = om;
96      [v, om, poses, stuck_timer, rand_goal] = robotNav3(ranges,
       base_angle, r_pose, poses, stuck_timer, rand_goal, goal, vp,
       dt);


98
        %%% MOTION NOISE %%%
100     v = v + .2*rand(1);
        om = om + .01*rand(1);
102
        %%% MOTION DAMPENING %%%
104     v = 0.1*v + 0.9*vp;
        om = 0.1*om + 0.9*omp;
106
```

```
    %%% MOTION MODEL %%%
108     r_pose(3) = r_pose(3) + om*dt;
        r_pose(1) = r_pose(1) + v*cos(r_pose(3));
110     r_pose(2) = r_pose(2) + v*sin(r_pose(3));

112     pause(1/256);
    end
```

./code/matlab/course_sim.m

A.2 Map

```
1   function [field_walls] = generateFieldWalls(max_dim)

3   %%% Field Walls are set up here as a list of line segments %%%
    % Wall structure [x1,y1;x2,y2] end points of line segment
5   outer_wall1 = [0,0;0,max_dim];
    outer_wall2 = [0,0;max_dim,0];
7   outer_wall3 = [0,max_dim;max_dim,max_dim];
    outer_wall4 = [max_dim,0;max_dim,max_dim];
9   outer_walls = [outer_wall1;outer_wall2;outer_wall3;outer_wall4];

11  island_wall1 = [202,137;202,202];
    island_wall2 = [118,137;118,202];
13  island_wall3 = [118,202;202,202];
    island_wall4 = [164,137;202,137];
15  island_walls = [island_wall1;island_wall2;island_wall3;
        island_wall4];

17  lr_wall1 = [118,91;200,91];
    lr_wall2 = [118,45;118,0];
19  lr_walls = [lr_wall1;lr_wall2];

21  ll_wall1 = [0,103;72,103];
    ll_wall2 = [72,103;72,46];
23  ll_walls = [ll_wall1;ll_wall2];

25  ur_wall1 = [72,157;72,max_dim];
    ur_wall2 = [46,157;72,157];
27  ur_walls = [ur_wall1;ur_wall2];

29  field_walls = 2*[outer_walls;island_walls;lr_walls;ll_walls;
        ur_walls];
```

./code/matlab/generateFieldWalls.m

A.3 Robot Plotter

```
function plotRobot(T, diameter, heading_length, color)
radius = diameter/2;
% Outline
for dth = 0:0.1:2*pi
        rx1 = radius*cos(dth);
        ry1 = radius*sin(dth);
        rx2 = radius*cos(dth + 0.1);
        ry2 = radius*sin(dth + 0.1);

        p1 = T*[rx1;ry1;1];
        p2 = T*[rx2;ry2;1];

        line([p1(1),p2(1)],[p1(2),p2(2)],'Color',color);
end

% Heading Line
line([T(1,3),heading_length*T(1,1) + T(1,3)],[T(2,3),
    heading_length*T(2,1) + T(2,3)], 'Color', color);
```

./code/matlab/plotRobot.m

A.4 Navigation

```
%%% Robot Navigation Algorithm %%%
% Input:
%     ranges    A 1D arrary of scalar distance measurements
% Output:
%     v         Robot linear velocity in cm/sec
%     om        Robot angular velocity in radians/sec
function [v, om, poses, stuck_timer, rand_goal] = robotNav3(
    ranges, base_angle, r_pose, poses, stuck_timer, rand_goal,
    goal, vp, dt)

% Constants
plot_scaler = 1000;
```

```
OM_MAX = 1;
VAR_THRESH = 5;


% Calculate Occlusion Forces
f_ranges = OccForce3(ranges, base_angle + 20);

% Calculate Goal Force
goal_range = sqrt((r_pose(1) - goal(1))^2 + (r_pose(2) - goal(2)
    )^2);
f_goal = (1/((0.5*min(goal_range,25000))^2))*[goal(1) - r_pose
    (1), goal(2) - r_pose(2)]';
f_goal = [cos(-r_pose(3)),-sin(-r_pose(3));sin(-r_pose(3)),cos(-
    r_pose(3))]*f_goal;

% Check for being Stuck
[stuck_timer, rand_goal] = CheckStuck(poses, VAR_THRESH,
    stuck_timer, rand_goal);

if((stuck_timer.time > 0) && (goal_range > 50))
    goal = rand_goal;
    goal_range = sqrt((r_pose(1) - goal(1))^2 + (r_pose(2) -
    goal(2))^2);
    f_goal = (1/((0.5*min(goal_range,25000))^2))*[goal(1) -
    r_pose(1), goal(2) - r_pose(2)]';
    f_goal = [cos(-r_pose(3)),-sin(-r_pose(3));sin(-r_pose(3)),
    cos(-r_pose(3))]*f_goal;
    stuck_timer.time = stuck_timer.time - 1
    if(stuck_timer.time == 0)
        stuck_timer.stuck = 0;
    end
end

% Calculate Inertia
f_heading = 0.01*[cos(0),sin(0)]';

% Sum Forces
f = f_ranges + f_goal + f_heading;




% Calculate Angular Velocity
th_in = (atan2(f(2),f(1)));
TURN_MAGNITUDE = 0.8;
```

```
49 om = TURN_MAGNITUDE*th_in*dt;

51 % Calculate Linear Velocity
   v_desired = 0.75*log(min(ranges)/30)*(5*exp(-om));
53 v = 0.1*vp + 0.9*v_desired;

55 % Angular Saturation
   if(om > OM_MAX)
57     om = OM_MAX;
   elseif (om < -OM_MAX)
59     om = -OM_MAX;
   end
61
   % Stop at goal
63 if(goal_range < 10)
       v = 0;
65     om = 0;
   end
67
   % Plot Forces
69 plotForce(f_ranges, r_pose, plot_scaler, 'g');
   plotForce(f_goal, r_pose, plot_scaler, 'm');
71 plotForce(f_heading, r_pose, plot_scaler, 'k');
   plotForce(f, r_pose, plot_scaler, 'b');
73

75 % In Graph Text
   text(10,6,'om: ');
77 text(70, 6, num2str(om));

79 % Update Poses
   poses = updatePoses(poses, v, om, dt);
```

./code/matlab/robotNav3.m

A.5 Occlusion Force Generator

```
1 function f_ranges = OccForce(ranges, base_angle)
  RANGE_OFF = 20;
3 RANGE_COEFF = 0.1;

5 % Right Sensor
```

```
f_range1 = -1/(RANGE_COEFF*((ranges(1) - RANGE_OFF)^2))*[cos(-
    base_angle),sin(-base_angle)]';

% Left Sensor
f_range2 = -1/(RANGE_COEFF*((ranges(2) - RANGE_OFF)^2))*[cos(
    base_angle),sin(base_angle)]';

f_ranges = f_range1 + f_range2;
```

./code/matlab/OccForce3.m

A.6 Force Plotter

```
function plotForce(occForce, r_pose, plot_scaler, color)
fr1 = [cos(r_pose(3)),-sin(r_pose(3));sin(r_pose(3)),cos(r_pose
    (3))]*occForce;
line([r_pose(1), plot_scaler*fr1(1) + r_pose(1)],[r_pose(2),
    plot_scaler*fr1(2) + r_pose(2)], 'Color',color);
```

./code/matlab/plotForce.m

A.7 Trap Detector

```
function [stuck_timer, rand_goal] = CheckStuck(poses, VAR_THRESH
    , stuck_timer, rand_goal)

pose_var = [var(poses(1,:)), var(poses(2,:))]';
if((pose_var(1) < VAR_THRESH) && ~any(poses(1,:) == 0) && (
    stuck_timer.stuck == 0))
    rand_goal = 500*rand(2,1);
    stuck_timer.time = stuck_timer.top;
    stuck_timer.count = stuck_timer.count + 1;
    stuck_timer = stuck_timer.updateTop();
    stuck_timer.stuck = 1;
    disp('BANG X');
elseif ((pose_var(2) < VAR_THRESH) && ~any(poses(2,:) == 0) && (
    stuck_timer.stuck == 0))
    rand_goal = 500*rand(2,1);
    stuck_timer.time = stuck_timer.top;
    stuck_timer.count = stuck_timer.count + 1;
    stuck_timer = stuck_timer.updateTop();
```

```
    stuck_timer.stuck = 1;
    disp('BANG Y');
end
```

./code/matlab/CheckStuck.m

A.8 Stuck Timer Class

```
classdef StuckTimer
    %UNTITLED7 Summary of this class goes here
    %   Detailed explanation goes here

    properties
        stuck = 0;
        time = 0;
        count = 0;
        top = 100;
    end

    methods
        function obj = updateTop(obj)
            obj.top = obj.count*100;
        end
    end

end
```

./code/matlab/StuckTimer.m

A.9 Sensor Model

```
%%% Beam Model %%%
function [range, feature] = getRangeBeam(map, view_angle,
    max_range, min_range, T)

    ranges = [];
    feature_iters = [];

    for iter = 1:2:length(map)
        % Parameterized Inclusion Test
        p_0 = T\[map(:,iter);1];
```

```
        p_f = T\[map(:,iter+1);1];

    %%%% DOT PRODUCT TEST %%%%
        p_1 = [p_0(1),p_0(2)]';
        p_2 = [p_f(1),p_f(2)]';
        p_lam = p_2 - p_1;
        distances = [];
        % Perpendicular Test
        % Calculate perpendicular point
        lambda_perp = (-p_lam'*p_1)/(p_lam'*p_lam);
        p_r = p_1 + lambda_perp*p_lam;
        pp1 = p_r(1)/sqrt(p_r(1)^2 + p_r(2)^2);
        % Check to see if perpendicular point is in the view
    angle and
        % the point is on the line segment
        if((pp1 >= cos(0.5*view_angle))&&((lambda_perp <= 1)&&(
    lambda_perp >= 0)))
            % Add distance to range candidates
            ranges = [ranges, sqrt(p_r(1)^2 + p_r(2)^2)];
            % Add feature iterator to feature iterator
    candidates
            feature_iters = [feature_iters, iter];
        else
            % End Point Test
            % Calculate the unit vector projection along the
    endpoint line
            % and see if it is within the view angle
            ep1 = p_1(1)/sqrt(p_1(1)^2 + p_1(2)^2);
            ep2 = p_2(1)/sqrt(p_2(1)^2 + p_2(2)^2);
            if(ep1 >= cos(0.5*view_angle))
                distances = [distances, sqrt(p_1(1)^2 + p_1(2)
    ^2)];
            end
            if(ep2 >= cos(0.5*view_angle))
                distances = [distances, sqrt(p_2(1)^2 + p_2(2)
    ^2)];
            end
            % Intersection Test
            % Calculate the intersection point and see if it is
    on the line
            % segment
            a = (p_lam(1)^2)*(cos(0.5*view_angle)^2 - 1) + p_lam
    (2)^2;
            b = 2*((p_1(1)*p_lam(1))*(cos(0.5*view_angle)^2 - 1)
     + p_1(2)*p_lam(2));
```

```
            c = (p_1(1)^2)*(cos(0.5*view_angle)^2 - 1) + p_1(2)
    ^2;
            % Check for imaginary solutions
            if((b^2 - 4*a*c) >= 0)
                lambdas_int = roots([a,b,c]);
                % If the solution is within range
                if((lambdas_int(1) <= 1)&&(lambdas_int(1) >= 0))
                    ip1 = p_1 + lambdas_int(1)*p_lam;
                    nrmip1 = ip1(1)/sqrt(ip1(1)^2 + ip1(2)^2);
                    % If the point is within the view angle
                    if(nrmip1 >= cos(0.5*view_angle))
                        distances = [distances, sqrt(ip1(1)^2 +
    ip1(2)^2)];
                    end
                end
                if((lambdas_int(2) <= 1)&&(lambdas_int(2) >= 0))
                    ip2 = p_1 + lambdas_int(2)*p_lam;
                    nrmip2 = ip2(1)/sqrt(ip2(1)^2 + ip2(2)^2);
                    % If the point is within the view angle
                    if(nrmip2 >= cos(0.5*view_angle))
                        distances = [distances, sqrt(ip2(1)^2 +
    ip2(2)^2)];
                    end
                end
            end
            % If any distances were added
            if(~isempty(distances))
                % Add distance to range candidates
                ranges = [ranges, min(distances)];
                % Add feature iterator to feature iterator
    candidates
                feature_iters = [feature_iters, iter];
            end
        end
    end

    [range, index] = min(ranges);
    Limit beam range
     if(range > max_range)
         range = max_range;
     end
     if(range < min_range)
         range = min_range;
     end
    iter = feature_iters(index);
```

```
87      feature = [map(:,iter),map(:,iter+1)];
```

./code/matlab/getRangeBeam.m

A.10 Sonar Beam Plotter

```
1  function drawBeam(view_angle, max_range, range, T, color)

3  % Transformed Beam Edges

5      BEAM_X = max_range*cos(0.5*view_angle);
       BEAM_Y = max_range*sin(0.5*view_angle);
7      beam_ep = zeros(3,2);
       beam_ep(:,1) = [BEAM_X,  BEAM_Y,  1]';
9      beam_ep(:,2) = [BEAM_X, -BEAM_Y,  1]';
       beam_epT = zeros(3,2);
11
       beam_epT(:,1) = T*beam_ep(:,1);
13     beam_epT(:,2) = T*beam_ep(:,2);

15     line([T(1,3),beam_epT(1,1)],[T(2,3),beam_epT(2,1)],'Color',
       color);
       line([T(1,3),beam_epT(1,2)],[T(2,3),beam_epT(2,2)],'Color',
       color);
17
       for dth = (-0.5*view_angle):0.05:(0.5*view_angle)
19         rx1 = range*cos(dth);
           ry1 = range*sin(dth);
21         rx2 = range*cos(dth + 0.025);
           ry2 = range*sin(dth + 0.025);
23
           p1 = T*[rx1;ry1;1];
25         p2 = T*[rx2;ry2;1];

27         line([p1(1),p2(1)],[p1(2),p2(2)],'Color',color);
       end
```

./code/matlab/drawBeam.m

A.11 Pose Estimator

```
function poses = updatePoses(poses, v, om, dt)

    r_pose = [0,0,0]';
    r_pose(3) = poses(3,1) + om*dt;
    r_pose(1) = poses(1,1) + v*cos(r_pose(3));
    r_pose(2) = poses(2,1) + v*sin(r_pose(3));

    poses = [r_pose, poses(:, 1:length(poses) - 1)];
    %x = [var(poses(1,:)), var(poses(2,:))]'
```

./code/matlab/updatePoses.m

Appendix B

Nao Code

B.1 Main Code

```
#include <iostream>
#include <alerror/alerror.h>
#include <alproxies/almotionproxy.h>
#include <alproxies/dcmproxy.h>
#include <alproxies/almemoryproxy.h>
#include <alproxies/alsonarproxy.h>
#include <alproxies/alsensorsproxy.h>
#include <alproxies/alredballtrackerproxy.h>
#include <qi/os.hpp>
#include <math.h>
#include <time.h>

#include "robotNav.h"

bool robotHalted = false;
double time_from_start = 0;

int main(int argc, char* argv[]) {
  if(argc != 2){
    std::cerr << "Wrong number of arguments!" << std::endl;
    std::cerr << "Usage: RApath NAO_IP" << std::endl;
    exit(2);
  }

  // Initialize Sonar and Memory Proxies to start Sonar
  // Hardware and retrieve data
  AL::ALSonarProxy sonarPrx(argv[1], 9559);
```

```
AL::ALMemoryProxy memPrxSonar(argv[1], 9559);

sonarPrx.subscribe("RAPlanner");
int period = sonarPrx.getMyPeriod("RAPlanner");

// Initialize Bumper and Memory Proxies
AL::ALSensorsProxy headTouchPrx(argv[1], 9559);
AL::ALMemoryProxy memPrxBumper(argv[1], 9559);

headTouchPrx.subscribe("RAPlanner");
float headTouched = memPrxBumper.getData("FrontTactilTouched")
  ;

// Values to store current distance measurements
AL::ALValue rightDist, leftDist;
float rightD, leftD;
// Sonar Value keystrings (locations in memory)
const std::string keyR = "Device/SubDeviceList/US/Right/Sensor
  /Value";
const std::string keyL = "Device/SubDeviceList/US/Left/Sensor/
  Value";

// Initialize Motion Proxy
AL::ALMotionProxy motionPrx(argv[1], 9559);
// Walk Velocity Variables
float Vx = 0;
float Vy = 0;
float Om = 0;
float stepFreq = 1;

// Pose
bool useSensorValues = true;
std::vector<float> pose = motionPrx.getRobotPosition(
  useSensorValues);
std::vector<float> headAngle = motionPrx.getAngles("HeadYaw",
  true);

// Set Head
motionPrx.setStiffnesses("Head", 1.0f);
motionPrx.setAngles("HeadYaw", 0.0f, 0.2f);
motionPrx.setAngles("HeadPitch", 0.0f, 0.2f);


// Initialize Red Ball Tracker
AL::ALRedBallTrackerProxy redBallPrx(argv[1], 9559);
```

```
  redBallPrx.startTracker();
69 std::vector<float> ballPose;
  float ballMag = 1000.0f;
71 int ballCounter = 0;
  ballPose.push_back(1000.0f);
73 ballPose.push_back(0.0f);
  ballPose.push_back(0.0f);
75
  // Initialize Walk
77 motionPrx.walkInit();
  motionPrx.setWalkTargetVelocity(Vx, Vy, Om, stepFreq);
79
  // File Processing
81 FILE* fh = fopen("sonarLog1.txt", "w");

83 AL::DCMProxy dcm_proxy(argv[1], 9559);

85 int t1 = dcm_proxy.getTime(0);
  try {
87   std::cout << "Test" << std::endl;
     // Poll Sonars
89   while(true){
       // Head Pressed?
91     headTouched = memPrxBumper.getData("FrontTactilTouched");
       if(headTouched) robotHalted = true;
93
       if(!robotHalted){
95       int t2 = dcm_proxy.getTime(0);
         time_from_start = (t2-t1)/1000.0;
97       // Update Ball Pose
         if(redBallPrx.isNewData()){
99         ballPose = redBallPrx.getPosition();
           // Convert to Centimeters
101        ballPose[0] *= 100;
           ballPose[1] *= 100;
103        ballPose[2] *= 100;

105        ballMag = sqrt(pow(ballPose[0],2) + pow(ballPose[1],2)
    );
         }
107      else{
           motionPrx.setAngles("HeadYaw", 0.0f, 0.2f);
109        motionPrx.setAngles("HeadPitch", 0.0f, 0.2f);
         }
111      // Get Distances
```

```
        rightDist = memPrxSonar.getData(keyR);
113     leftDist = memPrxSonar.getData(keyL);
        // Convert to Centimeters
115     rightD = 100*float(rightDist);
        leftD  = 100*float(leftDist);
117
        robotNav(ballPose, Om, Vx, rightD, leftD);
119
        if(ballMag < 30.0f){
121       if(ballCounter > 15){
            Vx = 0;
123         Om = 0;
            robotHalted = true;
125         std::cout << std::endl << " GOAL REACHED " << std::
    endl;
          }
127       ballCounter++;
          std::cout << " Ball Counter: " << ballCounter;
129     }

131     // Constrain velocities
        if(Vx < -1.0f) Vx = -1.0f;
133     if(Vx > 1.0f) Vx = 1.0f;
        if(Om < -1.0f) Om = -1.0f;
135     if(Om > 1.0f) Om = 1.0f;

137     // Update Walk
        motionPrx.setWalkTargetVelocity(Vx, Vy, Om, stepFreq);
139
        // Update Pose
141     pose = motionPrx.getRobotPosition(useSensorValues);

143     // Get Head Yaw
        headAngle = motionPrx.getAngles("HeadYaw", true);
145
        // Print Values
147     std::cout << " LDist: " << leftD;
        std::cout << " RDist: " << rightD;
149     //std::cout << " Vx: " << Vx << " Om: " << Om;
        //std::cout << " Period (ms): ";
151     //std::cout << " Pose: " << pose;
        //std::cout << " Ball Pose ";
153     //std::cout << " x: " << ballPose[0];
        //std::cout << " y: " << ballPose[1];
155     //std::cout << " z: " << ballPose[2];
```

```
            std::cout << std::endl;
157
            // Log sonar data
159         fprintf(fh, "%lf %lf %lf %lf %lf %lf\n", time_from_start
        , leftD, rightD, headAngle, ballPose[0],ballPose[1],ballPose
        [2]);

161       }
          else{
163         std::cout << "Robot Halted." << std::endl;
            motionPrx.setWalkTargetVelocity(0.0f, 0.0f, 0.0f, 0.0f);
165         redBallPrx.stopTracker();
            motionPrx.setStiffnesses("Head", 0.0f);
167         qi::os::sleep(30);
            fclose(fh);
169       }
        }
171
    }
173 catch (const AL::ALError& e) {
      motionPrx.setWalkTargetVelocity(0.0f, 0.0f, 0.0f, 0.0f);
175     std::cerr << "Caught exception: " << e.what() << std::endl
      ;
       exit(1);
177 }
  exit(0);
179 }
```

./code/nao/main.cpp

B.2 Robot Navigation Header

```
1 #ifndef _ROBOT_NAV_H
  #define _ROBOT_NAV_H
3
  #include <math.h>
5 #include <vector>
  #include <deque>
7 #include <iostream>

9 #define M_PI 3.14159265359

11 class Force{
```

```
public:
  float x;
  float y;
};


void robotNav(std::vector<float> &pose, float &Om, float &V,
    float rightD, float leftD);

Force OccForce(float rightD, float leftD, float base_angle,
    float sub_angle, float goal_range);

bool stuck(float V);

#endif
```

./code/nao/robotNav.h

B.3 Robot Navigation Implementation

```
#include "robotNav.h"

void robotNav(std::vector<float> &goal_pose, float &Om, float &V
    , float rightD, float leftD){

  // Constants
  float plot_scaler = 1000;
  float OMMAX = 1;
  float sub_angle = 15.0*(M_PI/180);
  float VAR_THRESH = 5;
  float base_angle = 45.0*(M_PI/180);

  // Memory
  static float Vp = 0;
  static float Omp = 0;
  static double prev_time = 0;

  // Calculate Goal Force
  float goal_range = sqrt(pow(goal_pose[0],2) + pow(goal_pose
    [1],2));
  Force f_goal;
  float goalCoeff = 250*(0.0001+pow(goal_range,-2.2f)+0.03*pow(
    goal_range,-1.0f));
```

```
22    f_goal.x = goalCoeff*(goal_pose[0]);
      f_goal.y = goalCoeff*(goal_pose[1]);
24
      //if (0)
26    //{
      //std::cout << " Goal Vector ";
28    //std::cout << " mag: " << goalCoeff;
      //std::cout << " ang: " << atan2(f_goal.y, f_goal.x);
30    //}
      //std::cout << std::endl;
32
      // Calculate Occlusion Forces
34    Force f_ranges = OccForce(rightD, leftD, base_angle, sub_angle
        , goal_range);

36    // Calculate Inertia
      Force f_heading;
38    f_heading.x = 0.01*cos(0.0);
      f_heading.y = 0.01*sin(0.0);
40
      // Sum Forces
42    Force f;
      f.x = f_ranges.x + f_goal.x + f_heading.x;
44    f.y = f_ranges.y + f_goal.y + f_heading.y;

46    //std::cout << " Force Vector ";
      //std::cout << " mag: " << sqrt(pow(f.y,2) + pow(f.x,2));
48    //std::cout << " ang: " << atan2(f.y, f.x);
      //std::cout << std::endl;
50
      // Calculate Angular Velocity
52    Om = 4.0*atan2(f.y,f.x)*dt;

54    // Calculate Linear Velocity
      float Vd = 0.4*log(1+0.2*std::min(leftD, rightD)/30)*(5*exp(-
        Om));
56    V = 0.1*Vp + 0.9*Vd;

58    Vp = V;
      Omp = Om;
60
}
62
```

```
Force OccForce(float rightD, float leftD, float base_angle,
    float sub_angle, float goal_range){
  float RANGE_OFF = 15.0;
  float RANGE_COEFF = -2400;
  if (goal_range < 75) RANGE_COEFF *= (goal_range/75.);
  double d_left = leftD_deriv->d , d_right = rightD_deriv->d;

  float rightCoeff = RANGE_COEFF*(pow((rightD - RANGE_OFF),-2)
    +0.15*pow((rightD - RANGE_OFF),-1.0f));
  float leftCoeff =  RANGE_COEFF*(pow((leftD - RANGE_OFF),-2)
    +0.15*pow((leftD - RANGE_OFF),-1.0f));

  Force f_range1, f_range2, f_ranges;

  // Right Sensor
  f_range1.x = rightCoeff*cos(-base_angle);
  f_range1.y = rightCoeff*sin(-base_angle);

  // Left Sensor
  f_range2.x = leftCoeff*cos(base_angle);
  f_range2.y = leftCoeff*sin(base_angle);

  f_ranges.x = f_range1.x + f_range2.x;
  f_ranges.y = f_range1.y + f_range2.y;

  return f_ranges;
}

bool stuck(float V){
  std::deque<float> pLinVels;
  float sum = 0;
  float avg = 0;
  // Keep a sliding window of velocities
  pLinVels.push_back(V);
  if(pLinVels.size() > 100) pLinVels.pop_front();

  // Take average
  for(size_t ndx = 0; ndx < pLinVels.size(); ndx++){
    sum += pLinVels[ndx];
  }
  avg = sum/pLinVels.size();

  if(avg < 0.2) return true;

  return false;
```

```
}
```

./code/nao/robotNav.cpp

Bibliography

[1] Wade E., Dye J., Mead R., & Matar M., *Assessing the Quality and Quantity of Social Interaction in a Socially Assistive Robot–Guided Therapeutic Setting* in IEEE International Conference on Rehabilitation Robotics (Zurich, Switzerland), June 2011

[2] Knight H., Satkin S., Ramakrishna V., & Divvala S., *A Saavy Robot Standup Comic: Online Learning through Audience Tracking* in International Conference on Tangible and Embedded Interaction (Funchal, Portugal), January 2011

[3] Feil-Seifer D., & Matarić M., *Human-Robot Interaction*. Invited contribution to Encyclopedia of Complexity and Systems Science, Meyers R. (eds.), Springer New York, pp. 4643–4659 (2009)

[4] DARPA Robotics Challege Home. Retrieved from `http://theroboticschallenge.org/`

[5] Muecke K. J., (2009). *An Analytical Motion Filter for Humanoid Robots* (Doctoral dissertation). Retrieved from `http://scholar.lib.vt.edu/theses/available/etd-04072009-183346/unrestricted/01-Body.pdf`

[6] Graf C., Hartl A., Röfer T., & Laue T., *A Robust Closed-Loop Gait for the Standard Platform League Humanoid*, in Proceedings of the Fourth Workshop on Humanoid Soccer Robots in conjunction with the 2009 IEEE-RAS International Conference on Humanoid Robots, Zhou C., Pagello E., Menegatti E., Behnke S., & Rofer T. (eds.), (Paris, France) pp. 30 37 (2009)

[7] Browatzki B., Tikhanoff V., Metta G., Bülthoff H., & Wallraven C., *Active Object Recognition on a Humanoid Robot* in IEEE International

Conference on Robotics and Automation (Saint Paul, MN), May 2012, pp. 2021–2028

[8] Wang Y., Lin M., & Ju R., (2010). *Visual SLAM and Moving-object Detection for a Small-size Humanoid Robot*, International Journal of Advanced Robotic Systems, Kordic V., Lazinica A., & Merdan M. (eds.), ISBN: 1729-8806, InTech, DOI: 10.5772/9700. Available from: `http://www.intechopen.com/journals/international_journal_of_advanced_robotic_systems/visual-slam-and-moving-object-detection-for-a-small-size-humanoid-robot`

[9] Gutmann J., Fukuchi M., & Fujita M., *Real-time path planning for humanoid robot navigation*, in Proceedings of the International Joint Conference on Artificial Intelligence, (2005), pp. 1232–1237

[10] Berenson D., Abbeel P., & Goldberg K., *A Robot Path Planning Framework that Learns from Experenice* in Proceedings from IEEE International Conference on Robotics and Automation, (Saint Paul, MN), May 2012, pp. 3671–3678

[11] Dudekm, G. & Jenkin, M. (2010). Representing and Reasoning About Space. In *Computational Principles of Mobile Robotics* (2nd ed., pp 167-210). Cambridge University Press.

[12] Stramandinoli F., Cangelosi A., & Marocco D., *Towards the grounding of abstract words: A Neural Network model for cognitive robots* in Proceedings of the International Joint Conference on Neural Networks, (San Jose, CA), July 2011, pp. 467–474

[13] Rusu R., Holzbach A., Diankovl R., Bradskil G., & Beetz M., *Perception for Mobile Manipulation and Grasping using Active Stereo* in Proceedings of the IEEE-RAS International Conference on Humanoid Robots, (Paris, France), Dec. 2009, pp. 632–638

[14] Pratt G. & Williamson M., *Series Elastic Actuators* in Proceedings of the International Conference on Robotic Systems, 1995, pp. 399–406

[15] Brooks G., Krishnamurthy P., Khorrami F., *Humanoid Robot Navigation and Obstacle Avoidance in Unknown Environments* in Proceedings of the Asian Control Conference, (Istanbul, Turkey) June 2013

[16] P. Krishnamurthy and F. Khorrami, "GODZILA: A low-resource algorithm for path planning in unknown environments," *Jour. of Intelligent and Robotic Systems*, vol. 48, no. 3, pp. 357–373, March 2007.

[17] Home - Corporate - Aldebaran Robotics—Accueil. Retrieved from `http://www.aldebaran-robotics.com/en/`

[18] Locomotion control. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/naoqi/motion/control-walk.html#control-walk`

[19] Locomotion control API. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/naoqi/motion/control-walk-api.html#control-walk-api`

[20] Joint control. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/naoqi/motion/control-joint.html`

[21] NAOqi Framework. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/dev/naoqi/index.html`

[22] qiBuild documentation. qiBuild 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/qibuild/index.html`

[23] Marshall W., *Basics of Robot Sonar* Retrieved from `http://www.wgmconsulting.com/Sonar.pdf`

[24] ALMemory. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/naoqi/core/almemory.html`

[25] Sonars. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/family/robots/sonar_robot.html`

[26] ALSonar - Getting Started. NAO Software 1.14.3 documentation. Retrieved from `http://www.aldebaran-robotics.com/documentation/naoqi/sensors/alsonar.html?highlight=sonar#getting-started`

[27] ALSonar API. NAO Software 1.14.3 documentation. Retrieved from https://community.aldebaran-robotics.com/doc/1-12/naoqi/sensors/alsonar-api.html#ALSonarProxy::subscribe__ssCR.iCR.floatCR

[28] Stiffness control. NAO Software 1.14.3 documentation. Retrieved from http://www.aldebaran-robotics.com/documentation/naoqi/motion/control-stiffness.html#control-stiffness

[29] NAOqi Trackers. NAO Software 1.14.3 documentation. Retrieved from http://www.aldebaran-robotics.com/documentation/naoqi/trackers/index.html