Thesis
Subtitle

Griswald Brooks

June 9, 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

List of Tables

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][10] 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.

For this project, the Nao humanoid platform by Aldebaran Robotics was used. 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 was 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. 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. Potential fields algorithms treat the mobile robot as a point mass that is repelled by obstacles and attracted by the goal.

The algorithm implemented was the Game-Theoretic Optimal Deformable Zone with Inertia and Local Approach (GODZILA) algorithm. 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 [11].

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

Path Planning Algorithm

The path planning algorithm implemented in this project was the Game-Theoretic Optimal Deformable Zone with Inertia and Local Approach (GODZILA) algorithm. This potential fields algorithm uses the closed form formulation of an optimized cost function that rewards motion towards a goal location and penalizes motion towards obstacles and in directions other than the current heading. In addition to the optimized cost function, GODZILA incorporates a straight line path planner when the goal is in sight of the robot and a randomization function to allow the robot to escape local minima.

Using the closed form formulation of the cost function in quadratic terms, the algorithm effectively generates forces that push the robot from obstacles and pull it towards the goal. The component penalizing motion in directions other than the current heading of the robot acts like an inertia term which helps prevent oscillations and forces any limit cycles to be larger, which helps eliminate certain local minima.

Occlusion forces are generated through use of sensor data from the robot's distance sensors. Each distance reading is used to generate a force magnitude which is applied as a vector to the robot along the orientation of the sensor relative to the robot. The goal force is generated in the same manner, with estimated distance to the goal generating a magnitude applied to the vector along the estimated orientation of the goal relative to the robot. The inertia term provides a constant bias towards the current heading of the robot.

The angle from the sum of these forces is then taken as the desired heading of the robot and converted into an angular rate with is passed to the vehicle.

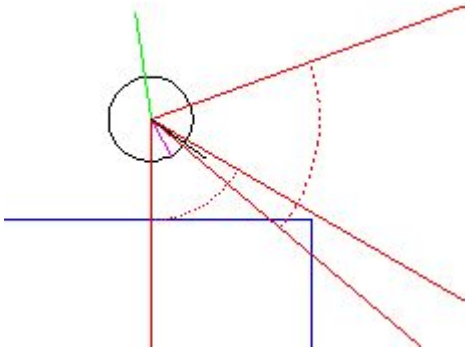

Figure 2.1: Simulation showing component forces. The green line shows the summed occlusion vectors, magenta is the goal vector, black is the inertia vector.

Chapter 3

Platform

The platform utilized on this project was 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 allowed 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 suffered from multipath returns and specular reflections which caused the robot to read incorrect distances at

Figure 3.1: Nao robot at Control/Robotics Research Lab

certain angles. For example, at times when the Nao was oriented toward a wall at an obtuse angle, the right distance sensor, which was closer to the wall, would 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 stemmed from the material that the occlusion was made from. It was found that if the material was too soft, the distance reported to it would be larger, or at times, not be seen. Harder materials seemed to correct for this. Soft materials were items like plastic trash cans. Metal trash cans responded more reliably. A brief overview about various issues with sonar can be read about in [17].

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

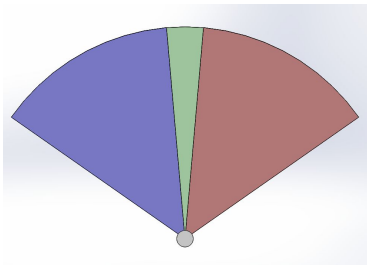

Figure 3.2: Nao sonar cones. The left cone is shown in blue and the right cone is shown in red. The green cone shows the overlap between the two.

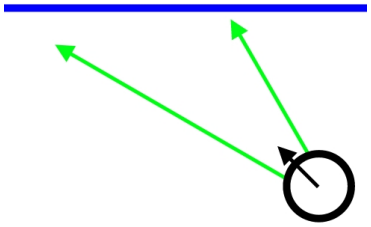

Figure 3.3: Distance measurement error at obtuse angles. In the figure, the black circle represents the Nao in the plane, while black arrow represents the current heading. The blue line represents a wall and the green arrows indicate the direction the sonars are pointing. When in positions similar to 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[15]. 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 was not compiled on the Nao itself but rather on an external computer and commands were sent to the Nao via a Wi-Fi network that both the Nao and the external computer were connected to. After setting up the build system qiBuild [16] 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 [18]. 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 [19].

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.[20][21]

```
   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 [13]. 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 [12]. The Nao can be stopped by setting all of the target velocities to zero.

For goal estimation, the inbuilt red ball tracker API was 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 [23]. 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 [22]. The "Head" is a kinematic chain which refers to a collection of joints, in this case the yaw and pitch axes of the head [14]. 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 **??**

Chapter 4

Simulation

Algorithm development and robot simulation was done in MATLAB. The simulation constructed a virtual environment using line segments and approximated the robot as a point on the plane. The kinematic model of the robot was taken as a differential drive robot, as modeling the full kinematics of the Nao robot was not necessary to the efficacy of the algorithm. A basic approximation for inertia was made by low pass filtering all velocity commands. Motion noise was injected by adding uniform random noise between zero and one, which was then scaled by a constant, to the linear and angular velocities. The resultant velocities were integrated to gain robot position using RK4 integration. The sonar beams were 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 were also perturbed by uniform random noise before being returned to the navigation algorithm.

Chapter 5

Results

Appendix A

Code

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] 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

[11] 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

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

[13] 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

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

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

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

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

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

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

[20] 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

[21] 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

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

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