OMNI-DIRECTIONAL FORCE-FEEDBACK FOR ASSISTED NAVIGATION OF OMNI-DIRECTIONAL ROBOTS

by

Rajat Tyagi

A thesis submitted to the faculty of
The University of Utah
in partial fulfilment of the requirement for the degree of

Master of Science

Department of Mechanical Engineering

The University of Utah

May 2017

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Rajat Tyagi**

has been approved by following supervisory committee members:

Stephen A. Mascaro	, Chair	**12/20/2016** Date Approved
Sanford G. Meek	, Member	**12/20/2016** Date Approved
David E. Johnson	, Member	**12/20/2016** Date Approved

and by **Timothy A. Ameel** , chair of

the department of **Mechanical Engineering**

and by David B. Kieda , Dean of The Graduate School.

ABSTRACT

The objective of this research is to improve the ability of a human operator to drive an omnidirectional robot by using omnidirectional force-feedback. Omnidirectional vehicles offer improved mobility over conventional vehicles and can potentially benefit people requiring motorized transportation and industries where vehicles must operate in confined spaces. However, omnidirectional vehicles require more skill to control due to the additional degrees of freedom inherent in the vehicle's design. We hypothesize that providing force-feedback to the driver through an omnidirectional joystick will allow the robot to assist the driver in navigating and avoiding collisions with obstacles in a manner that is natural to the operator. This research is the first attempt to use true omnidirectional 3-DOF (degree of freedom) force-feedback to provide navigational assistance for a human to drive an omnidirectional vehicle. While 2-DOF force-feedback has been used in a limited capacity for obstacle avoidance on omnidirectional vehicles, this is the first study to include a third rotational axis of force-feedback and use it to guide a driver along planar collision-avoiding trajectories with a natural coordination of orientation. Unique intellectual merits put forth by this research include use of a novel omnidirectional haptic device and force-feedback strategies to guide operators and experiments to quantify the ability of force-feedback to improve omnidirectional driving performance and driver experience in a real time scenario.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Stephen Mascaro for his expert guidance, encouragement, and mentorship. Thanks to all my colleagues and lab mates who have put up with me during this time.

I would also like to thank Dr. Sanford Meek and Dr. David Johnson for being on my committee and providing input on my work.

Most of all, I would like to thank my parents for their love, pastience and support. They set my feet on this path by encouraging me to pursue my interests. Special thanks to my sisters; without their assistance I would have never made it through.

CHAPTER 1

INTRODUCTION

1.1 Motivation

The objective of this research is to improve the ability of a human operator to drive an omnidirectional robot by using omnidirectional force-feedback. Omnidirectional vehicles offer improved mobility over conventional vehicles and can potentially benefit people requiring motorized transportation and industries where vehicles must operate in confined spaces. However, they require more skill to control the additional degrees of freedom inherent in the vehicle's design. We hypothesize that by providing force-feedback to the driver through an omnidirectional joystick, the omnidirectional robot can assist the driver's navigation such that targets are reached and collisions with obstacles are avoided in an intuitive and efficient manner. Omnidirectional vehicles are becoming more common as their unique mobility advantages are being exploited in an increasing number of industries. Well-suited for confined spaces, they can be found in places such as loading docks and warehouses, where space is limited and highly mobile loading equipment is beneficial. A few examples of current production vehicles used in such scenarios include the ODV (Omni Directional Vehicle) produced by Hammond Technical Services Inc.[1], which can be used for a variety of tasks such as loading, towing, and plowing. Due to its inherent zero turning radius, the driver can maintain any desired orientation while performing a stationary task. A second example is the Airtrax Sidewinder (Figure 1.1), a large scale forklift built by Vehicle Technologies, Inc. The entire vehicle maneuvers on an omnidirectional platform, a highly desirable characteristic for a large and mobile loading vehicle operating in often congested loading areas [2]. Another example is the Kuka omniMove (Figure 1.2), manufactured by Kuka Robotics, which is a universal transport vehicle that can lift and move heavy planes [3]. Omni directional vehicles can also be used in urban search and rescue operations as scouts and in military as autonomous bomb-sensing vehicles and personal robotic assistants.

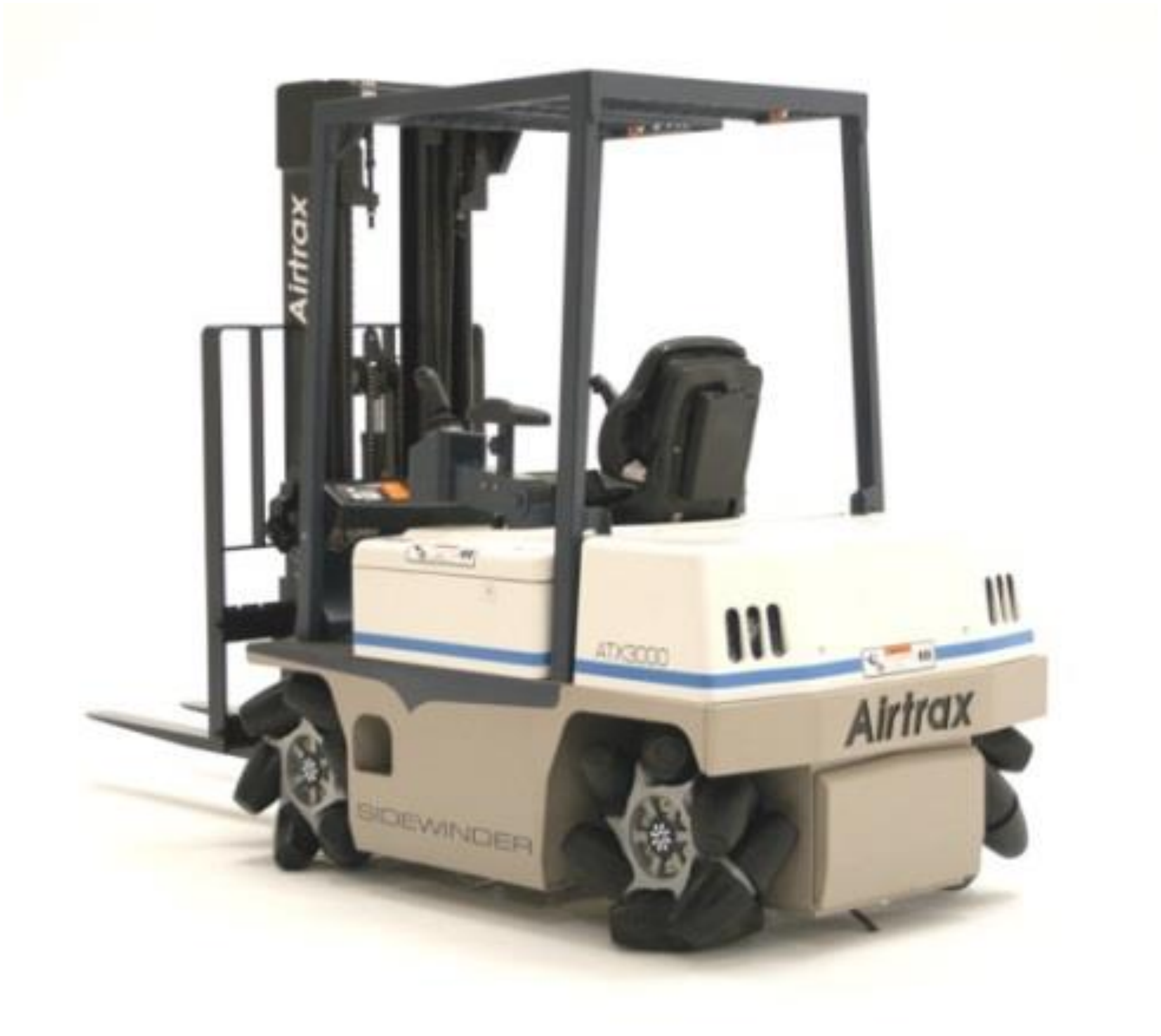


Figure 1.1: Airtrax Sidewinder, manufactured by Vetex.

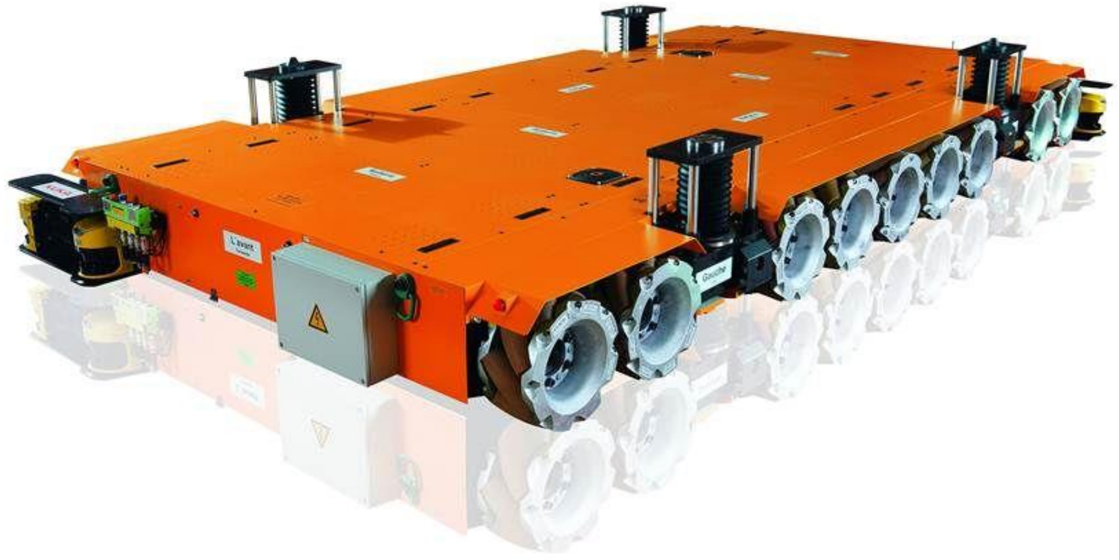


Figure 1.2: A Kuka omniMove omnidirectional universal transport vehicle, manufactured by Kuka Robotics.

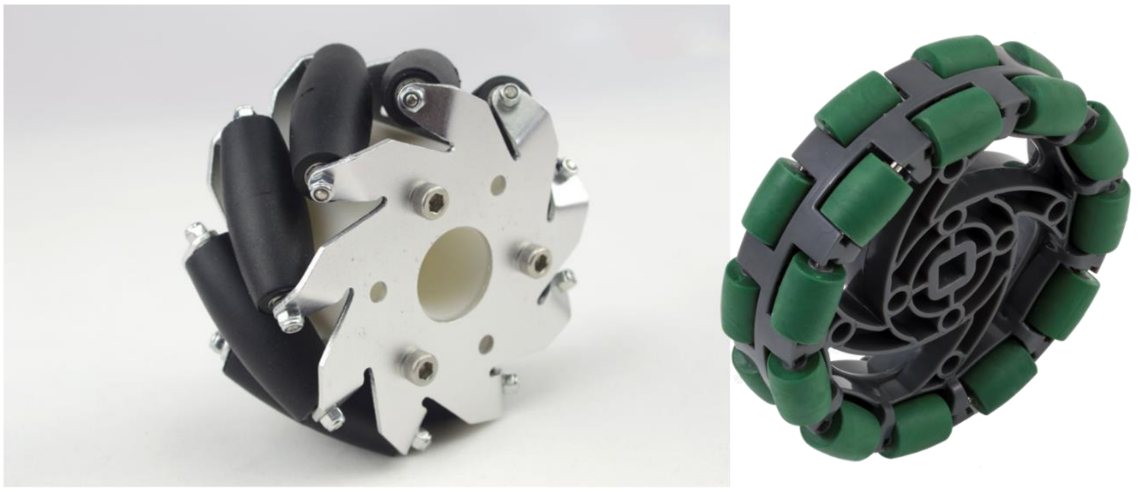
While some tabe suitable for autonomous navigation, it is often required or desirable for practical tasks that a human operator teleoperate the robot. Several factors can make the use of an omnidirectional robot difficult, such as embodiment, difficulty of control, and unintuitiveness of the three degrees of freedom. When operating robotic vehicles, haptic feedback can greatly increase the amount of information that can be intuitively conveyed to the user and improve the ability to safely and efficiently navigate a particular path. However, commercially available force-feedback joysticks almost exclusively apply force-feedback in the longitudinal and lateral directions only, and in the case of omnidirectional robots, there may be a separate joystick to control the third degree of freedom. We hypothesize that control and embodiment of an omnidirectional robot can be significantly improved by providing intuitive omnidirectional haptic feedback in a single joystick, such that the degrees of freedom of the joystick directly correspond to the degrees of freedom of the omnidirectional robot.

This thesis offers a novel idea for control of omnidirectional robots that is expected to improve operator comfort and navigational performance, while allowing the driver to remain in control. To do this, we propose the use force-feedback. Our ultimate goal is to experimentally quantify the navigational assistance provided by force-feedback.

This research is the first attempt to use true omnidirectional 3-DOF (degree of freedom) force-feedback to provide navigational assistance for a human to drive an omnidirectional vehicle. While 2-DOF force-feedback has been used in a limited capacity for obstacle avoidance on omnidirectional vehicles, this would be the first research to include a third rotational axis of force-feedback and to use it to guide a driver along planar collision-avoiding trajectories with a natural coordination of orientation.

1.2 Omnidirectional Robots

Omnidirectional robots, unlike their conventional counterparts, are able to translate laterally and rotate while staying in place. Holonomic omnidirectional robots are able to simultaneously move in their three degrees of freedom in the plane of the floor. In other words, they are able to simultaneously translate in X, translate in Y, and rotate about the Z axis. The most prominent method used to produce holonomic omnidirectional motion involves the use of Mecanum wheels [4]-[11] Figure 1.3(a), or Omni wheels [12] Figure 1.3(b) both are variations of same concept, a wheel comprised of a disk with rolling elements. In both cases, the configuration of the rolling elements gives the wheels one active degree of freedom and one passive degree of freedom. Mecanum wheel-based platforms are robust and mechanically simple compared to other designs and are better at operating in uneven terrain.



(a) (b)

Figure 1.3: Commercially available wheels. (a) Mecanum wheel produced by Nexus-Robots. (b) Omni wheel developed by Vex Robotics.

The Mecanum wheel platforms are driven by a single motor per wheel and the desired trajectory is input in the form of a three dimensional velocity vector. The inverse kinematics given in equation 1.1 convert the desired omni robot velocities into the required wheel velocities through the use of an inverse Jacobian matrix, which is derived from the physical dimensions of the omni robot.

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = J^{-1} R_z{}^T(\theta) \begin{bmatrix} v_x \\ v_y \\ \dot{\theta} \end{bmatrix} \qquad \textbf{(1.1)}$$

Here, J is the velocity jacobian, $R_z(\theta)$ is the rotational matrix, w_i is the velocity of ith wheel and $\begin{bmatrix} v_x & v_y & \dot{\theta} \end{bmatrix}^T$ is the vehicle velocity.

One other mechanism to produce holonomic omnidirectional motion based on the works of West and Asada [13] and improved upon by Mascaro [14] is to use spherical balls as wheels. Their method of supporting the wheels with roller bearings enables each wheel to have one active and one passive degree of freedom [15]. The kinematics are similar to that of Mecanum wheels, and while the resulting motion of the ball-wheeled vehicle is smoother in comparison to Mecanum-wheeled vehicles, it has more difficulty navigating rough terrain.

1.3 Haptic Interface

The use of haptic feedback in control devices is becoming increasingly common, and can be found in familiar technologies like cellular phones, gaming controllers, and complex precision instruments such as teleoperated surgical tools [16]. There are many forms of 2- DOF joystick controllers which are commercially available, the most common ones use potentiometers [16], Hall Effect sensors [18], or optical encoders [19] to measure user input. While a standard joystick can be used for controlling an omnidirectional robot, it will typically require a second input device to control the third degree of freedom of holonomic motion. Integrating the translation and rotational controls for holonomic motion into a single controller [20] can be accomplished using a joystick with a twist grip [21] as is found on many joysticks in video game controllers. Gaming joysticks are widely available, inexpensive and have frequently been employed in omnidirectional wheelchair research [22]. However, no commercial joysticks offer force-feedback in the rotational degree of freedom and experience has shown that the gaming joysticks lack the power, resolution, and fidelity to accurately perform haptic rendering.

Haptic feedback devices serve to present a user with the physical sensation that mimics real world forces. These devices are often combined with visual displays to enhance the user’s awareness of a virtual created world [23][24] or to assist in a specific task [25][26]. Haptic feedback is being employed in an ever increasing range of operations from education [27][28] to automotive interfaces [29] and is particularly useful in situations requiring improved human control [30]. Research has shown that haptic guidance enhances operator motor learning especially in steering oriented tasks. Because of this, haptic feedback is being used to train young wheelchair-bound children [31] to drive their wheelchairs. The application of haptic force-feedback in omnidirectional robot control is a very active field of research [32]-[37], with large amounts of currently active work dedicated to omnidirectional robots.

Most commercially available haptic devices are expensive, thus, many universities have started developing their own version of haptic joystick. One of these is the haptic paddle developed by Provancher and Doxon (Figure 1.4) at the University of Utah. This is a low-cost 1-DOF haptic paddle which can generate forces up to 47N at the top of the handle. This design was inspired from the haptic paddles invented at Stanford University and Rice University [38][39]. All of these use a capstan drive mechanism to transmit forces generated from motors to the haptic paddle handle and Hall Effect sensors to measure its position. Another modification to improve functionality to the one DOF haptic paddle was developed at the John Hopkins University, where they coupled two haptic paddles to form a 2 DOF haptic device called Snaptic Paddle. There are several other commercially available low-cost joysticks commonly used in research to drive various kind of robots. These are essentially manufactured for use in gaming controllers, but have been modified in-house for experimental research. The Logitech force pro is one these available joysticks and can provide force-feedback in 2 DOF, but these joysticks cannot produce a high amount of torque as demanded by various research. Other expensive commercially available haptic joysticks the Immersion Technologies' Impulse Stick and Impulse Engine that generate forces in the range of 8 to 14.5 N.

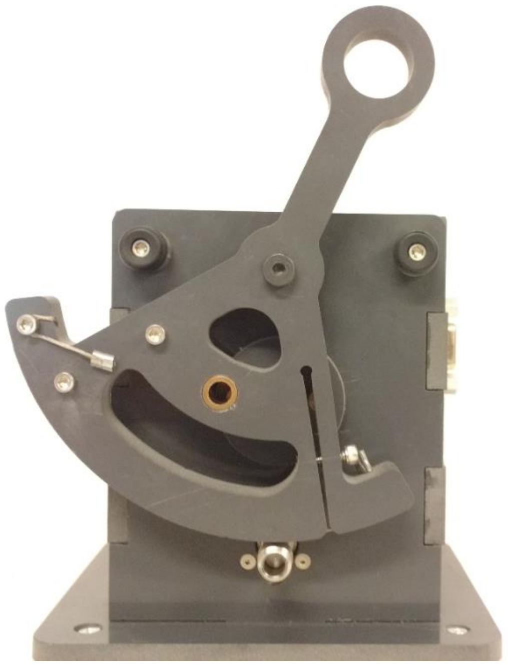


Figure 1.4: The University of Utah Haptic Paddle developed by Provancher and Doxon.

1.4 Compliance Center

One other important concept used in this research is the Compliance Center. In a classic peg in hole situation the location of compliance center plays an important role for proper alignment of a peg in a hole and avoids wedging or jamming [40]. The goal of Whitney's research was to describe rigid part mating during assembly which essentially is the assembly of parts that do not substantially deform. The author suggested that there are four stages of assembly – approach, chamfer crossing, one-point contact, and two-point contact. These events are shown in Figure 1.5.

The part rotates and translates during mating as there are usually initial lateral and angular errors between the parts that need to be corrected, hence, compliance support must be provided for at least one of the two assembling parts both laterally and in rotation. The compliance center is the point where all the forces are supposed to act so that the vehicle rotates in the direction that prevents wedging or jamming.

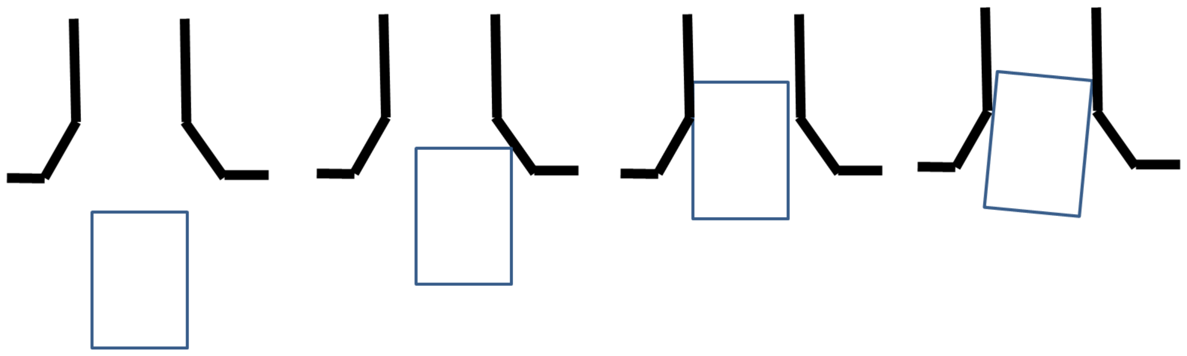


Figure 1.5: Stages of assembly: approach, chamfer crossing, one-point contact, two-point contact.

Whitney proved that the closer the compliance center is to the front of the part, the less force is required to mate two parts. This concept could also be used in robots to prevent jamming in confined spaces. So, only the rotational degree of freedom is dependent on the position of the compliance center, the translation degrees acts as before. In order to explore this concept further for use with omnidirectional robots, it was implemented on our robot and multiple tests were done by moving the compliance center at different spots.

1.5 Force-Feedback for Omnidirectional Wheelchairs

The most relevant study regarding this research was done by Urbano and Kitagawa at Toyohashi University of Technology and Gifu National College of Technology in Japan [41]. Their paper presented a haptic feedback control of a holonomic omnidirectional wheelchair with a haptic joystick for operation by disabled or elderly people. They developed their own holonomic omnidirectional mobile wheelchair for this research, comprising of three modes such as autonomous, semiautonomous, and power assist modes. Their wheelchair operates under these modes using ultrasonic and position sensitive device sensors for extracting environmental information. Their omnidirectional wheelchair has four wheels and all of them are driven by separate individual motors, with each wheel passively equipped with free rollers at circumference. The velocity of the omnidirectional wheelchair is the vector sum of velocities of the four Omni wheels. For providing haptic feedback, they designed a haptic joystick in which the desired velocity and moving direction of the omnidirectional wheelchair was proportional to the tilt angle and direction of the joystick. The impedance of the joystick was proportional to the distance to the obstacle, with a closer obstacle having higher impedance. However their control algorithm generated torques in only 2 DOF on the joystick, in only the X and Y direction with no rotating torque.

The desired dynamics equation of the joystick reference model of control counter-torque is given by:

$$\tau = J_d \ddot{q} + D_d \dot{q} + K_d q \qquad \textbf{(1.2)}$$

where τ and q are the torque and angular position of the joystick's motor, K_d, D_d and J_d are the joystick's desired stiffness, damping, and inertia, respectively.

This makes the real dynamics of the joystick:

$$T_r - \tau = J_a \ddot{q} + D_a \dot{q} + K_a q \qquad \textbf{(1.3)}$$

where K_a, D_a, and J_a are the joystick's physical stiffness, damping, and inertia respectively. Also, the desired inertia and damping of the joystick are assumed to be constant, which makes stiffness the only variable in the setup and is given by the equation:

$$K_d = K_o \cdot \left(\frac{\frac{v}{v_{max}} + \alpha}{\left(\frac{r}{r_{max}}\right)^2} + 1 \right) \qquad \textbf{(1.4)}$$

where, K_o is the initial value of stiffness, v is the omnidirectional wheelchair velocity, v_{max}

is the maximum velocity, r is the distance to the obstacle, r_{max} is maximum measurable distance of ultrasonic sensors, and α is constant which holds the effect of r when v is zero.

However, after preliminary trials they observed that there were problems of vibrations on the joystick when the vehicle was very close to obstacles, due to the non-linear behavior of the joystick. Thus, the authors decided to also vary the joystick's damping and inertia along with its stiffness. They chose the optimal values of stiffness, damping, and inertia coefficients through simulations. They then conducted simulations to test the effectiveness of their feedback algorithms and concluded that with the appropriate values of K_d, D_d, and J_d a smooth haptic counter-torque can be generated. However, all of their experiments were with feedback in just 1 DOF of the joystick with no evidence of simulation being conducted in haptic feedback in multiple DOF of the joystick.

This work was further extended by another group at Toyohashi University of Technology in Japan [42]. It is mostly based on a navigation guidance system for an omnidirectional wheelchair to navigate it through narrow spaces, such as elevator doors, using a haptic joystick. For this research, the authors used the same omnidirectional wheelchair that was used by previous groups at the same university. A similar experimental setup was used with two LIDAR sensors mounted at the front and back of the wheelchair to obtain information on the surroundings. For implementation purposes they considered the wheelchair as an eclipse and then generated another area called the recognition area, which is the area between two lines that are parallel to the path of the wheelchair motion and tangent to the eclipse vehicle area.

They designed a custom joystick for this purpose, which has one motor installed in both the x and y directions, which helps in generating haptic feedback in 2 DOF with a virtual spring-damper characteristic. The impedance of the joystick is based on the distance to the obstacle and wheelchair's linear velocity. The resultant feedback force can be represented by the equation:

$$\tau = D\dot{q} + Kq \qquad \textbf{(1.5)}$$

Here τ is the motor torque, K is the stiffness of the virtual spring, D is the virtual viscous damping coefficient, and q is the tilting angle of the joystick from the zero position. The virtual stiffness of the joystick is found by equation 1.4, where K_o is the initial value of stiffness, v is the omnidirectional wheelchair velocity, v_{max} is the maximum velocity, r is the distance to the obstacle, r_{max} is maximum distance in the effective range measured my LIDAR sensors, and α is a constant which depends on operator's characteristic of handling the wheelchair with the joystick. The velocity input is calculated by the position measured by the potentiometers mounted on the joystick in both directions, which is then converted to the velocity using a constant gain. The author's results seems promising but have haptic feedback in only 2 DOF, also there is no evidence of human trials done to check the effectiveness of their algorithm. The tests which were done by the author are very basic with only one obstacle in front of the robot and incudes no complex situations which were initially proposed in the research.

Another group at Toyohashi University of Technology in Japan [43], extended this work further. They presented a novel operational assistant system using laser scanning sensors in the power assist system using the handle with a 6-DOF force/torque sensor in order to induce operator simultaneously evading obstacle. They introduced a power assist system for the omnidirectional wheelchair in which a handle for power assist control is attached on the wheelchair with the 6-DOF force/torque sensor, which detects the added force and torque. This added force is transformed into the velocity reference by the first-order lag controller for power assist:

$$\begin{bmatrix} v_x \\ v_y \\ w \end{bmatrix} = \begin{bmatrix} \dfrac{K_{vx}}{T_{vx}\ s+1} & 0 & 0 \\ 0 & \dfrac{K_{vy}}{T_{vy}\ s+1} & 0 \\ 0 & 0 & \dfrac{K_w}{T_w\ s+1} \end{bmatrix} \begin{bmatrix} f_x \\ f_y \\ m \end{bmatrix} \qquad (\mathbf{1.6})$$

where $[v_x, v_y, w]$ are the reference velocity of the omnidirectional wheelchair, $[f_x, f_y, m]$ are the added forces to the handle by operator, $[K_{vx}, K_{vy}, K_w]$ are the gains of the power assist controller to transform the added forces into reference velocity, and $[T_{vx}, T_{vy}, T_w]$ is a gain parameter which depends on skill level of operator. They used two LIDAR sensors to map the environment on the back and sides of wheelchair, but not on the front side, due to the placement of sensors and the assumption that the operator takes account obstacles which are in front of the wheelchair. The operator's input force and torque is restricted by K_{vx} and K_{vy}, which also restricts the motion of the wheelchair. Another parameter is introduced in the algorithm called collision risk, which is essentially the risk of a collision on the path of robot and is given by the following equation:

$$k_i = 1 - \left(\frac{r_{min}}{r}\right)^k \qquad (\mathbf{1.7})$$

$$k = (v_{imax} - v_i\alpha)\beta, \qquad i = x, y$$

Here, r is the distance to the nearest obstacle on the path and v_i is the velocity input of the wheelchair. α and β are constants determined by the operability of the omnidirectional wheelchair. Then, they performed some brief simulation testing to check the effectiveness of the algorithm. In their simulation, the wheelchair advances towards a faraway wall with certain amount of force. The result shows that when the operator tries to move the wheelchair in the direction of the obstacles, the wheelchair decreases its speed by reducing the power gain.

One of the other more relevant works done in this field is by Fattouh, Anas, Mhamed Sahnoun, and Guy Bourhis [44]. They evaluated the use of a force-feedback joystick for a powered wheelchair. The joystick they used was a Microsoft Sidewinder force-feedback joystick and the factor used to determine the feedback force of the joystick were the displacement vectors of the 16 sensors to the nearest obstacles. The authors first tested their work in a virtual environment using a real time computer simulation, that give them possibility for testing different control algorithms, different force-feedback laws for the joystick and, at the end, different environment configurations. They then tested the final algorithm on an actual wheelchair equipped with sixteen ultrasonic sensors. The human operator applied a force on the input joystick in order to drive the wheelchair to the desired position. The joystick position is interpreted as the desired speed of the wheelchair and is then converted to the appropriate wheel velocities. The ultrasonic sensors read the direct distance to the obstacle in their detection range and an appropriate feedback force is generated on the joystick using the equations below:

$$F = \sum_{i=1}^{16} \alpha_i f_i \qquad \mathbf{(1.8)}$$

Here, α_i is a constant weight, f_i is the feedback force for the ith sensor which is determined by the equation:

$$f_i = \frac{1}{d_i}\, e^{j\,(\pi+\theta_i)} \qquad \mathbf{(1.9)}$$

where θ_i is the angle of the vector between the ith reading from the sensor and the obstacle and d_i is the magnitude of that vector. The test was performed on 7 subjects and the performance of each subject was evaluated based on of the number of collisions with obstacles and the time needed to complete the run and the total travelled distance. The mean of the results was calculated for analysis. The trial results show fewer collisions with the force-feedback algorithm and no significant difference in terms of distance travelled and the time to complete between the two driving modes; however, the wheelchair trajectory is smoother with the proposed force-feedback algorithm. They also noticed that the factors used for evaluation are not independent, as the time needed to complete a run depends on the number of collisions. In their future work, they propose to use some independent evaluation factor for better qualitative analysis. They also propose to test the above algorithm on an actual wheelchair on persons with and without disabilities. Although the collision results were promising the authors failed to provide any metric for the smoothness of trajectory of the wheelchair with and without force-feedback.

The limitation of all the above research was that none of them used any actual 3-DOF force-feedback. All the work in this area has been done with only 2-DOF haptic joysticks. The research in this thesis is the first to implement 3-DOF haptic feedback on a joystick for navigation of omnidirectional vehicles.

CHAPTER 2

EXPERIMENTAL SETUP

2.1 Joystick Design

In order to intuitively control the additional degrees of freedom inherent in omnidirectional vehicle's design, an omnidirectional haptic joystick is required. Due to the lack of commercially available 3 DOF haptic joysticks, it was decided to manufacture a custom-built haptic joystick with 3 DOF feedback. This haptic joystick was based on the joystick designed by Mascaro and Christensen at University of Utah [46], modifying it with an improved gear ratio and more powerful motors to generate higher torque. In this joystick, all the desired inputs are integrated in one device to control all 3 degrees of freedom of the omnidirectional robot simultaneously. The omnidirectional velocities are determined by the angular displacement of all three motors on the joystick. The angular displacements on the joystick are mapped to the corresponding linear and angular velocities of the omnidirectional robot.

The final joystick design was a modified version of the haptic joystick designed at the Bio Robotics lab, University of Utah, which in itself was a derivative of the 1-DOF haptic paddle designed by Provancher and Doxon at University of Utah, see Figure 2.1. This joystick was based on the capstan drive mechanism which helps with backlash-free driving, low slip, and smooth operation.

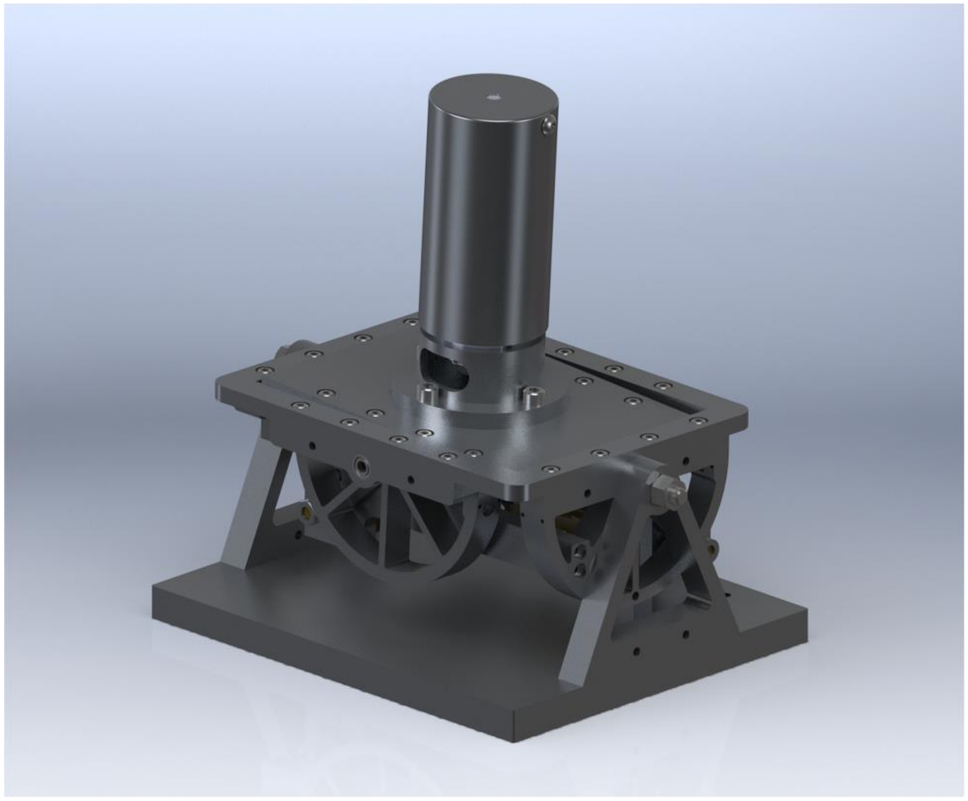


Figure: 2.1: Solid model of final 3-DOF haptic joystick design.

Each axis of the joystick is driven by a direct current (DC) motor with a 15:1 gear ratio capstan drive to provide necessary torque required to provide force-feedback at the joystick handle. This joystick was based on the capstan drive mechanism which helps with backlash free driving, low slip, and smooth operation. Each axis of the joystick is driven by a DC motor with a 15:1 gear ratio capstan drive to provide necessary torque required to provide force-feedback at the joystick handle. All the motors have attached incremental optical encoders for measuring the angular position of the joystick, which gets mapped to the linear and angular velocities of the robot. The motors used for this joystick are Maxon 310007 DC motors with a nominal torque of 85.6 mNm at the motor shaft, driven by AMC 30A8 servo amplifiers in current mode. Each motor is attached with a U.S. Digital E4P encoder with 360 counts per revolution to read the angular position of the joystick. This joystick can produce a force of 13.5 N at the center of joystick handle in both the X and Y axis and a rotational torque of 4.4 Nm.

A capstan drive mechanism was used to transmit torque from the motor shaft to the joystick handle, with capstan cable tensioned enough to maintain sufficient traction between capstan pulley and cable for rapid response to input torque. A nylon-coated stainless steel cable was used for the drive to reduce the wear and tear of cable and the pulley. The final finished haptic joystick used for the experiments is shown in Figure 2.2.

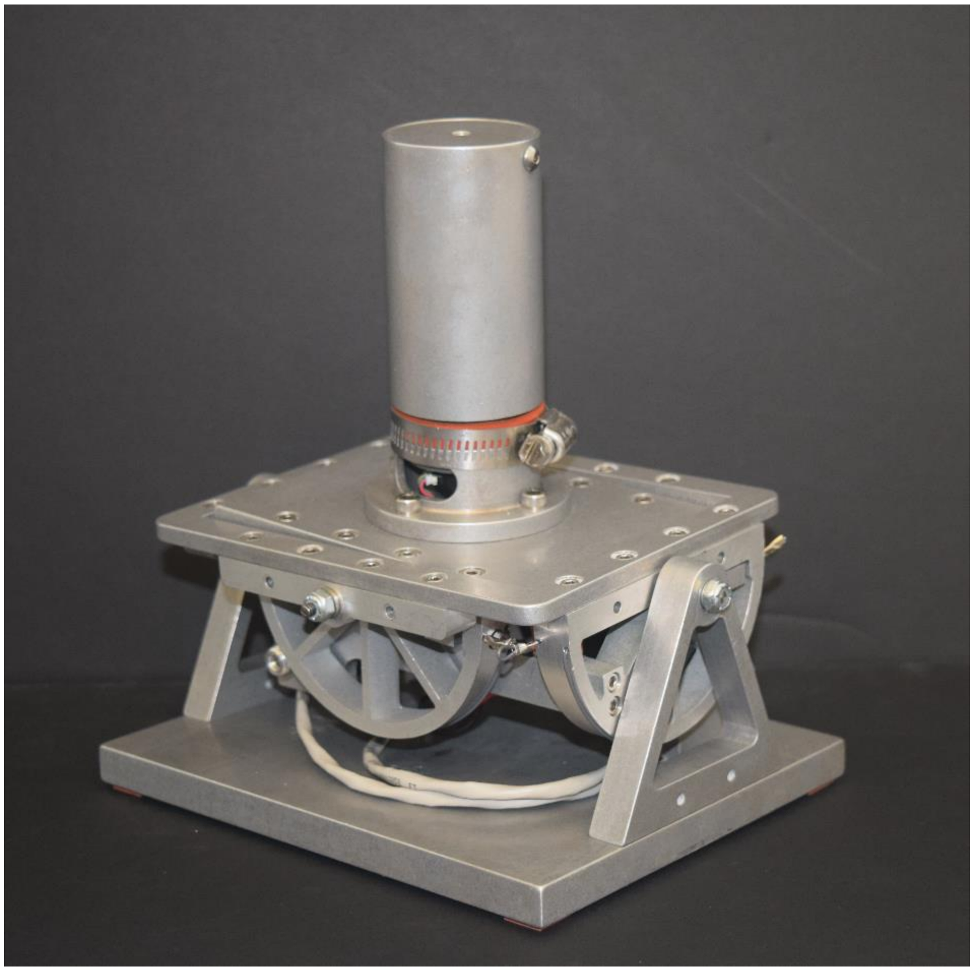


Figure 2.2: Final joystick.

2.2 Joystick Control System

As mentioned in the previous section, the above joystick has absolute incremental encoders to determine the tilt angle in the x and y directions and rotation in θ. But the joystick lacks any counterweights or springs to bring itself to the zero position or starting position. This makes it very hard to drive in 3 degrees of freedom because at any point in time the user has to provide the velocity input to the robot in all degrees, as well as bring the joystick at center to stop the robot or stop actuation in that direction. The joystick acts as a 2-DOF inverted pendulum as the handle of the joystick protrudes above the axes of robot. In order to let the joystick bring itself back to its home position, a proportional-derivative (PD) controller was implemented as shown in the Figure 2.3. ψ is the angular position of the joystick in all three degrees of freedom [ψ_x, ψ_y, ψ_θ]; where $\boldsymbol{e}$ is the error between zero position and the actual position of joystick. The P and D gains for the controller are selected using an iterative tuning process. Too high of a P gain makes the joystick too stiff to be used as a haptic feedback device and too low of a gain means the joystick will fail to bring itself to the zero position. An integral controller was not used as it would have resulted in an accumulated integral error over the time, and thus, a huge force-feedback when the joystick was maintained at a certain angular position corresponding to a velocity for long duration of time.

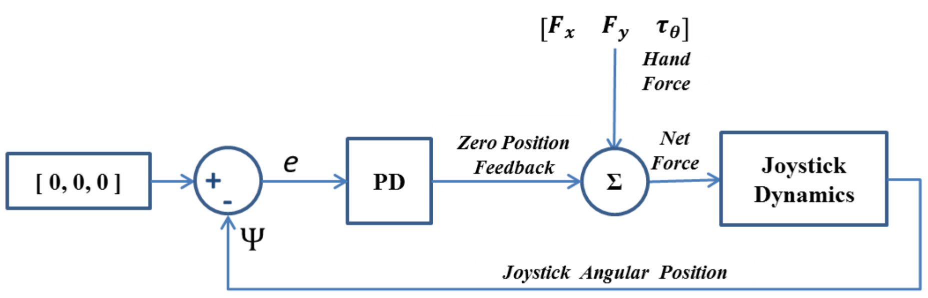


Figure 2.3: Joystick PD controller.

This problem was overcome using a gravity compensator, which reduced the effective PD gains required to bring the joystick to the zero position. This compensator counteracts toques due to gravity by treating all the eccentrically positioned movable components as a point load and calculates the effective torque due to gravity. Once this torque in calculated it is fed back to the joystick to compensate for the gravity. The modified control diagram with gravity compensation is shown in Figure 2.4, where τ' is the gravity-compensated torque. The gravity compensator was implemented in only the x and y directions as the rotation handle have all the components eccentric to its axis. The compensation was calculated by equation 2.1.

$$\tau = mgr\, sin(\psi) \tag{2.1}$$

As shown in Figure 2.5, m is the total mass of the motor, capstan mechanism, and joystick handle, ψ is the tilt angle from the center of joystick in that axis, r is the moment arm, and g is the acceleration due to gravity.

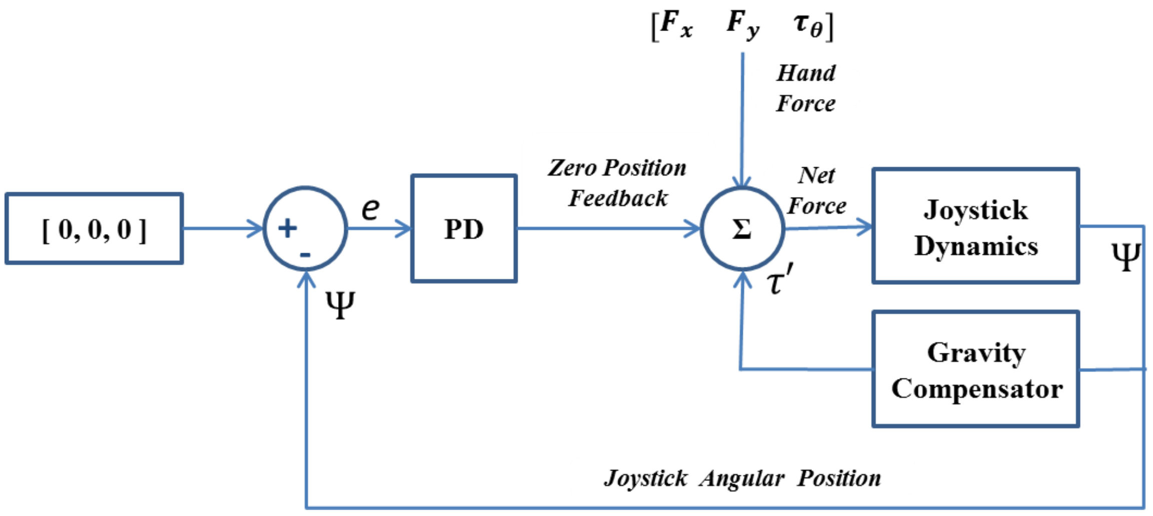


Figure 2.4: Joystick PD controller with gravity compensator.

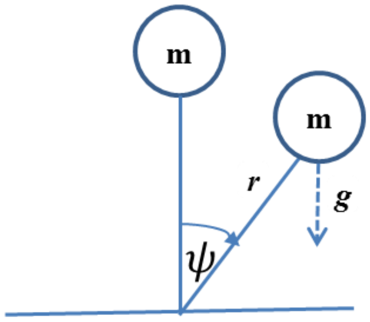


Figure 2.5: Inverted pendulum**.**

Implementing the PD controller with gravity compensation and a low value of PD gains successfully let the joystick bring itself to its zero position when no disturbance from the user's hand is provided. Gravity compensation eliminates the nonlinear response of the joystick due to gravity and thus the system can be treated as a linear system. The angular position of the joystick from the encoders is read by an Arduino microcontroller connected to the joystick using the quadrature output of the optical encoders. This microcontroller also provides the pulse-width-modulation (PWM) commands to the servo amplifier connected to the joystick motors to generate necessary force-feedback as determined by the feedback algorithm.

2.3 Omnidirectional Robot Hardware

The initial testing was done on an omnidirectional robot with Mecanum wheels, this robot is made by Nexus Robots (Figure 2.6). It has four DC Faulhaber motors with a voltage rating of 12 V and optical encoders to read the wheel position. The motors are controlled by motor driver boards provided by Nexus, and an Arduino microcontroller. 6 infrared sensors (IR) sensors were mounted on the robot to read the distance to the obstacles, as shown in Figure 2.7.

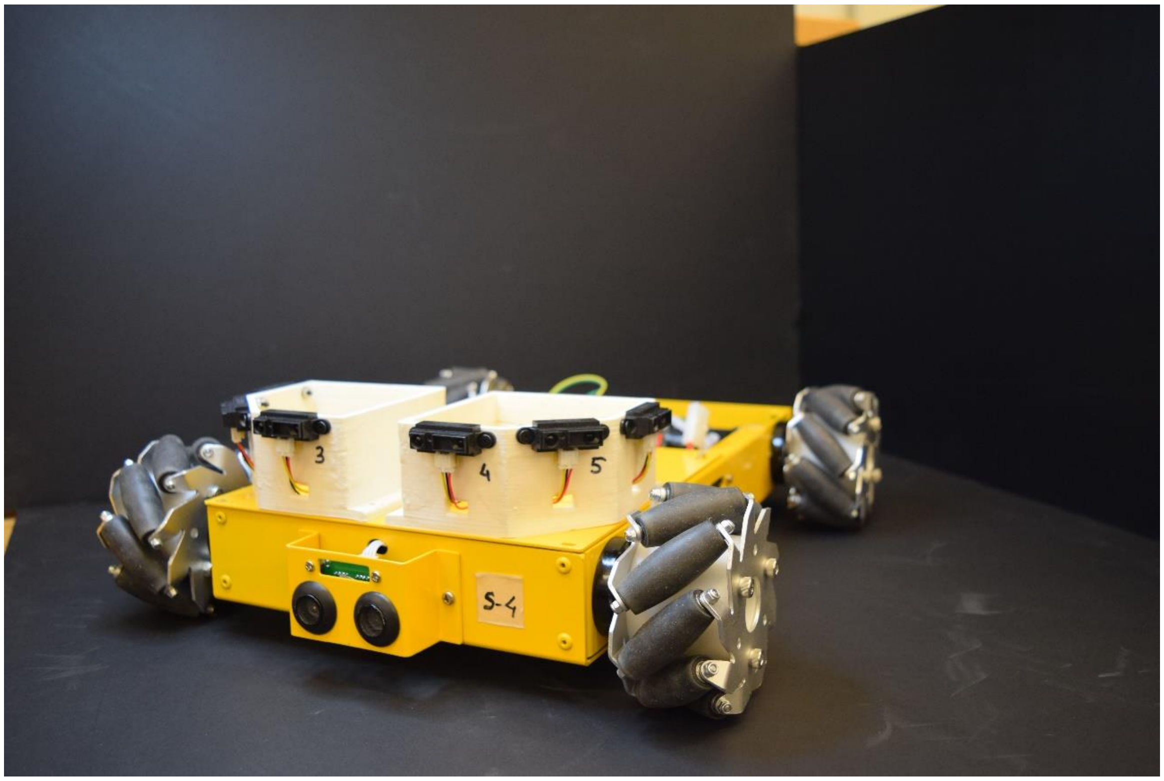


Figure 2.6: Omnidirectional robot.

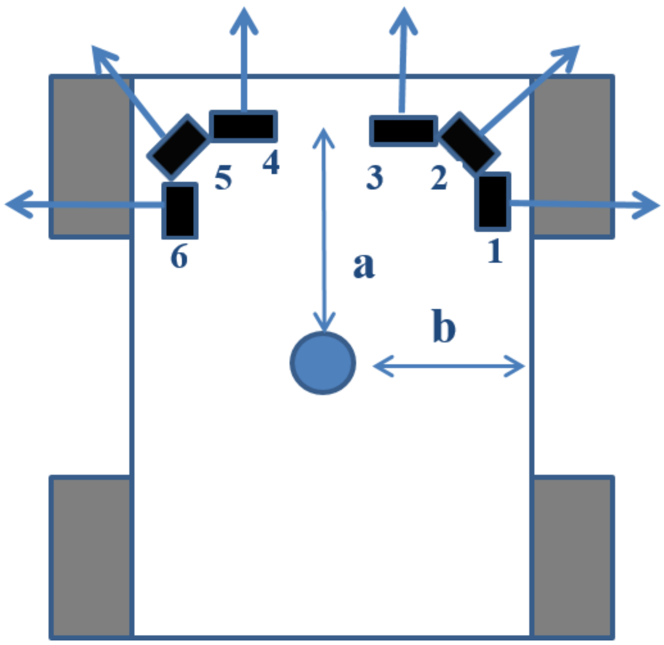


Figure 2.7: Array of IR sensors attached to the robot.

The omnidirectional wheel velocity is calculated from the joysticks angular positions using the inverse kinematics given in equation 2.2.

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = J^{-1} R_z{}^T(\theta) \begin{bmatrix} v_x \\ v_y \\ \dot{\theta} \end{bmatrix} \qquad \textbf{(2.2)}$$

Here, $\boldsymbol{J}$ is the Velocity Jacobian, $R_z(\theta)$ is the Rotational Matrix, w_i is the velocity of ith wheel, and $[v_x \quad v_y \quad \dot{\theta}]^T$ is the vehicle velocity. J^{-1} is given by the equation 2.3:

$$J^{-1} = \frac{1}{R_w} \begin{bmatrix} 1 & -1 & -l \\ 1 & 1 & -l \\ -1 & 1 & -l \\ -1 & -1 & -l \end{bmatrix} \qquad (2.3)$$

Here l is the length of the vehicle and R_w is the radius of the Mecanum wheel. The omnidirectional robot velocity was maintained using a PID controller, the complete control diagram is shown in Figure 2.8. Here Ψ is the angular position of the joystick $[\,\psi_x, \psi_y, \psi_\theta\,]$ and k_v is the gain to convert angular position of joystick to the X, Y, and angular velocities of the robot.

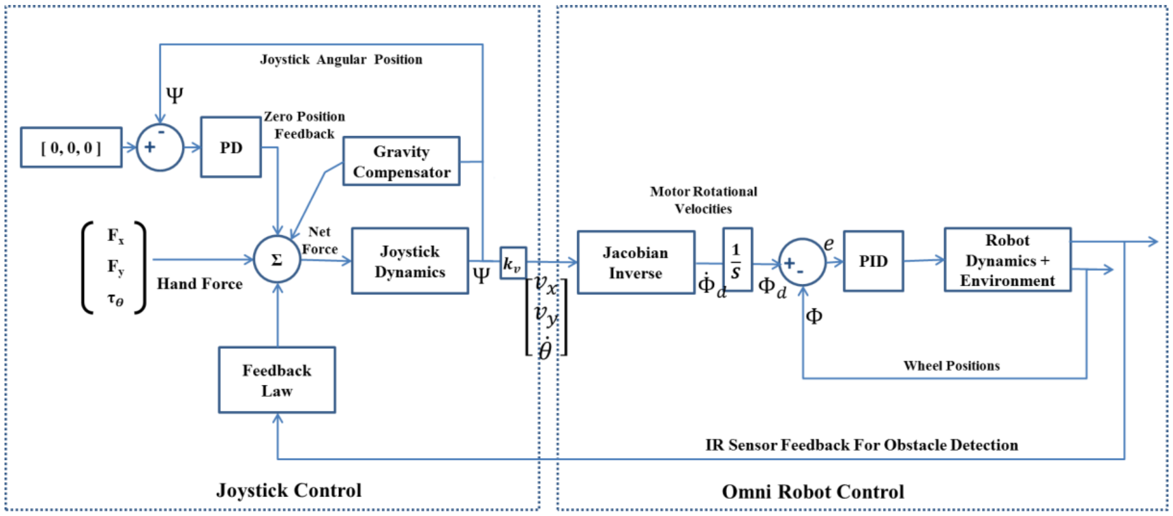


Figure 2.8: Complete block diagram with robot hardware.

The Jacobian inverse transforms these velocities into $\dot{\Phi}_d$, the desired motor velocities of robot, Φ is the actual wheel positions of robot and Φ_d is the desired position. e is the wheel position error which is to be penalized by the PID controller. The feedback law is a modified form of potential field which provides feedback as a virtual spring and virtual damper setup and is given by the equation 2.6.

$$K = \begin{bmatrix} 0 & -k\ sin\theta & -k & -k & k\ sin\theta & 0 \\ -k & -K\ cos\theta & 0 & 0 & k\ cos\theta & k \\ 0 & ak\ cos\theta - bk\ sin\theta & 0 & 0 & -ak\ cos\theta + bk\ sin\theta & 0 \end{bmatrix} \qquad (2.4)$$

$$B = \begin{bmatrix} 0 & -d\ sin\theta & -d & -d & d\ sin\theta & 0 \\ -d & -d\ cos\theta & 0 & 0 & d\ cos\theta & d \\ 0 & ad\ Cos\theta - bd\ Sin\theta & 0 & 0 & -ad\ Cos\theta + bd\ Sin\theta & 0 \end{bmatrix} \qquad (2.5)$$

$$\begin{bmatrix} F_x \\ F_y \\ \tau_\theta \end{bmatrix} = B\dot{x_s} + Kx_s \qquad (2.6)$$

Here k is the spring constant, d is the damping constant, x_s is the 6 x 1 array of distance measured from IR sensor, and $\dot{x_s}$ is a 6 x 1 array of relative velocities between the robot and the obstacle measured from the wheel velocities from the encoders. The feedback is also proportional to the velocity of robot, which provides feedback strongly when approaching obstacles rapidly and weakly when approaching obstacles slowly. The readings from IR range sensors were filtered by first averaging values and then implementing a low pass filter to further remove noise from the sensors. This algorithm was tested with the robot and the joystick in a real time scenario, shown in Figure 2.9. In this scenario, the user operates the robot from the start of the tunnel, traverses to the end, rotate 180°, and then comes back to the start position with force-feedback. After testing it was realized that there are vibrations in the joystick due to force-feedback caused by the discreteness of the force field, as there were only 6 IR range sensors to provide distance.

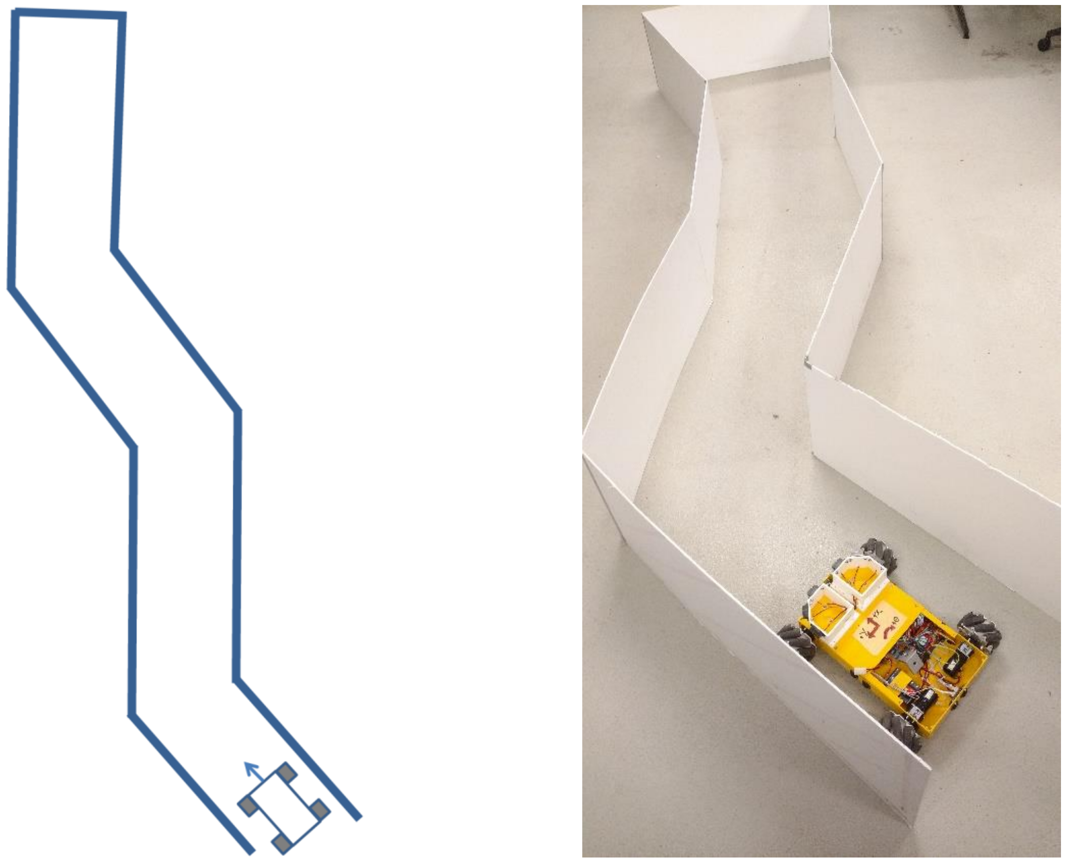


Figure 2.9: Testing feedback law with physical robot.

This vibration was amplified on the corners as the distance values switched from high to low or vice-versa in a matter of milliseconds, as illustrated in Figure 2.10. Also, since the sensors were in close proximity to each other, they caused interference with each other. This error is amplified as a result of the P gains used to generate force-feedback. The user misinterprets these vibrations as force-feedback, causing the control of robot to not be smooth and intuitive. Thus, it was realized that in order to generate smoother force-feedback, more sensors in closer proximity to generate more data points in the force field were required. However, this would also increase interference between the sensors, and thus, more vibrations on the joystick. It was obvious that some other sensor was needed which could provide more data points.

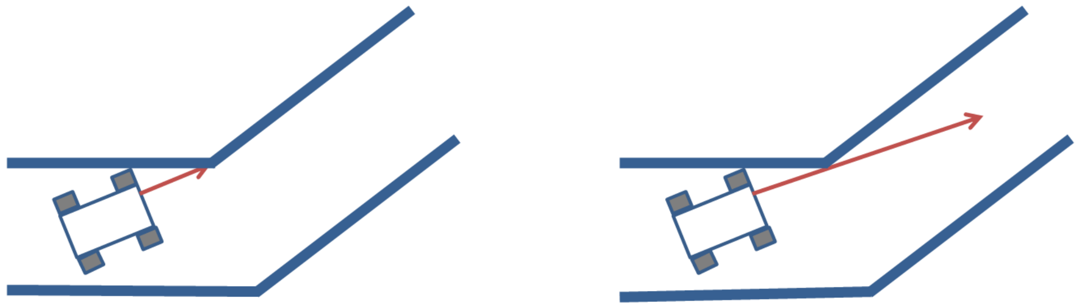


Figure 2.10: IR sensor reading fluctuation in the corner scenario.

2.4 Omnidirectional Robot Simulation

In order to get multiple data points of the obstacle distances, it was decided to use LIDAR sensors as they provide a complete environment scan with angular resolution as low as 0.25° and a scanning rate of 40 – 50 Hz. This would provide an almost continuous force field so as to generate fewer vibrations. However good LIDAR sensors are expensive, costing around $ 2500 to 5000 dollars per sensor, and at least two sensors were required to cover the full 360° environment. Also, processing that much data would require a higher quality onboard processor. In order to avoid such an investment, it was decided to move to simulations instead of using a real omnidirectional robot. There are many simulation software packages which provide real-time simulation for various robotic platforms. Virtual Robot Experimentation Platform (V-REP) made by Coppelia Robotics was used. This is a commercially available simulation software, which is free and open source for educational purposes. V-REP was used in ROS (Robot Operating System) on a computer running Ubuntu 14.04. The joystick is connected to the Arduino microcontroller as before with the Arduino sending the joystick's angular position to ROS using a ROS node which communicates the data to the V-REP simulation. For the V-REP model of the omnidirectional robot, we modified a preexisting Kuka's Youbot robot (Figure 2.11).

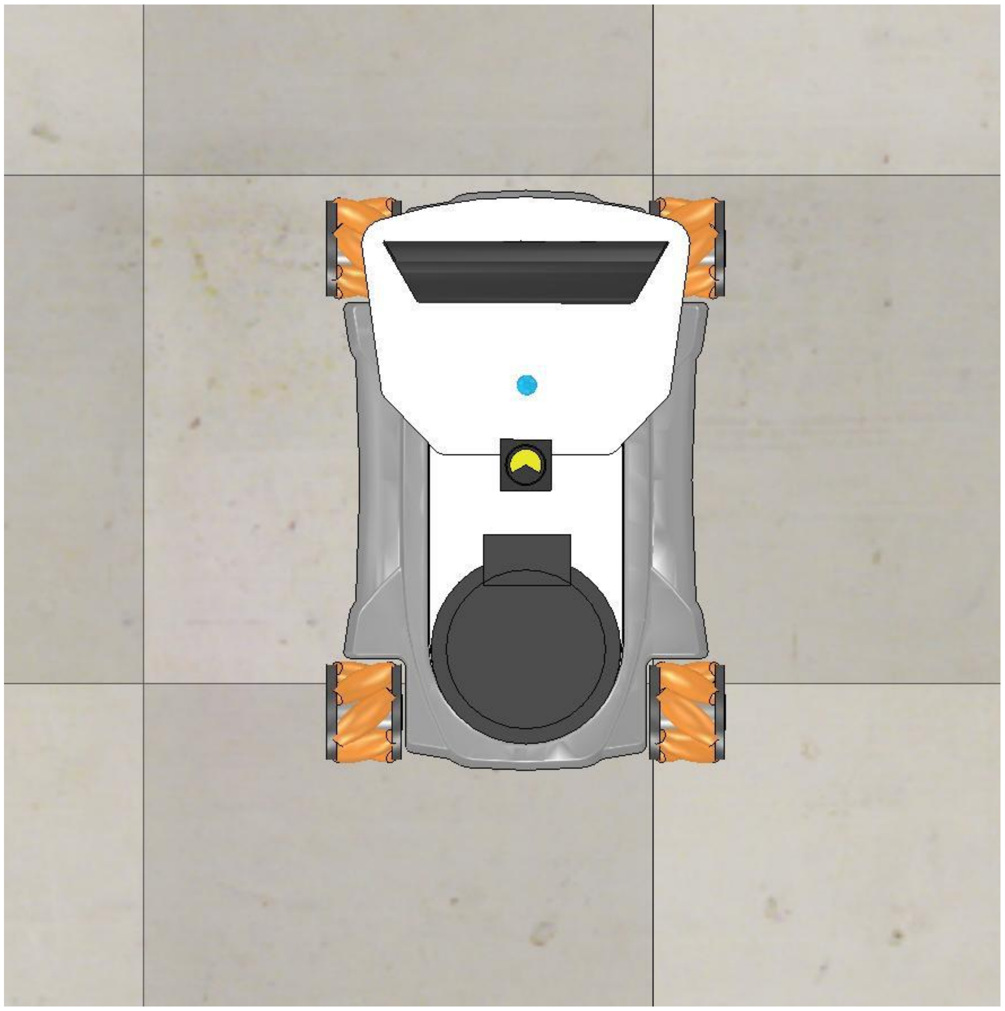


Figure 2.11: Model of modified Kuka's Youbot robot in V-REP.

The existing code which we used to run the real robot was modified to suit the robot in the simulation platform. Conceptually only the robot's velocity Jacobian was changed for the new robot model, as the length of Kuka's Youbot is bigger than the Nexus omnidirectional robot that was used before. The control law was changed accordingly, as there was no need of a PID controller to control the robot's velocity in the simulation. The new block diagram is shown below in Figure 2.12. It should be noted that only the robot side of control is changed for simulation, while the joystick's control is the same as before with no change whatsoever.

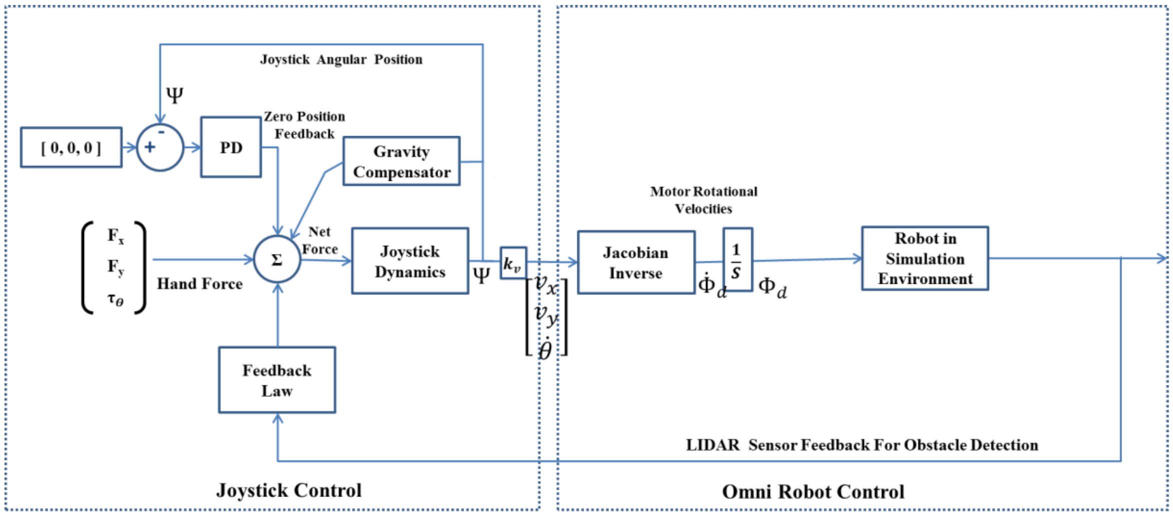


Figure 2.12: New block diagram for simulation.

A Hukoyu LIDAR sensor was placed at the center of the robot to create a uniform force field around the robot. The program script of the LIDAR sensor was modified to completely scan the 360° environment, and the angular resolution was increased to 2.8° to save computation time. The data obtained from the LIDAR Sensor are used to generate a circular force field originating at the center of robot, as shown in Figure 2.13 (left). A few primitive test scenarios were created in the simulation, and it was observed that the circular force field was creating instability in confined spaces as there was always some part of the force field which was in contact with the wall, see Figure 2.14. The spring force generated by a circular force field are not uniform with the rectangular footprint of robot, as the distance from the robot to the edge of the force field is not uniform.

Figure 2.15(a) shows the spring analogy of the force field. The spring force is at a maximum if there is any obstacle near the center axis of robot and at a minimum around the corner of the robot given the same amount of deflection. This created spikes in the force-feedback on the joystick. This effect was magnified on corners or narrow passages and created instability in the joystick, see Figure 2.16. This was overcome by creating a

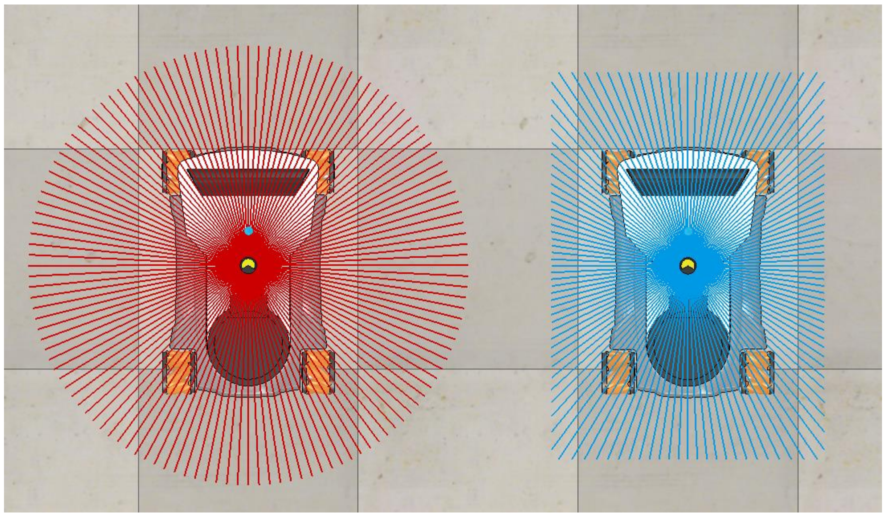


Figure 2.13: Circular (left) and rectangular (right) force field around robot.

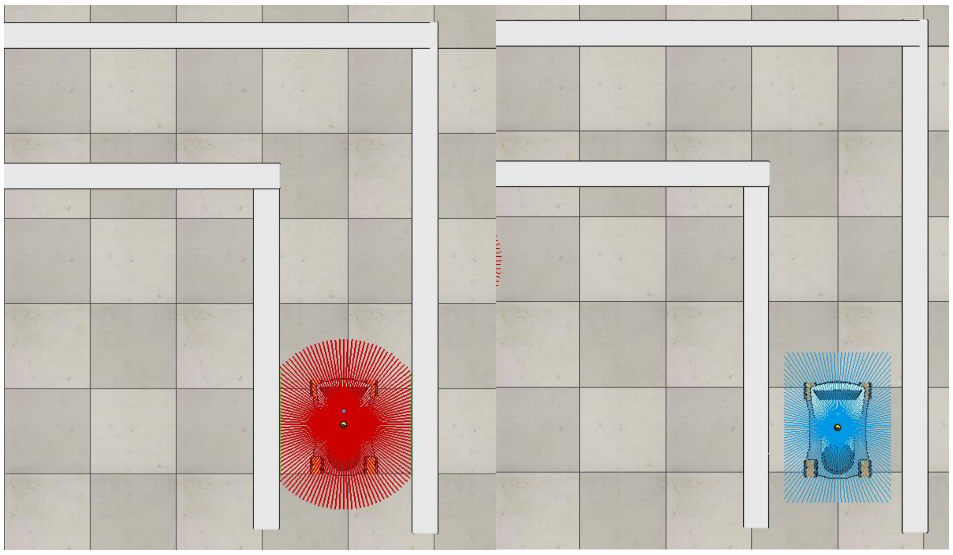
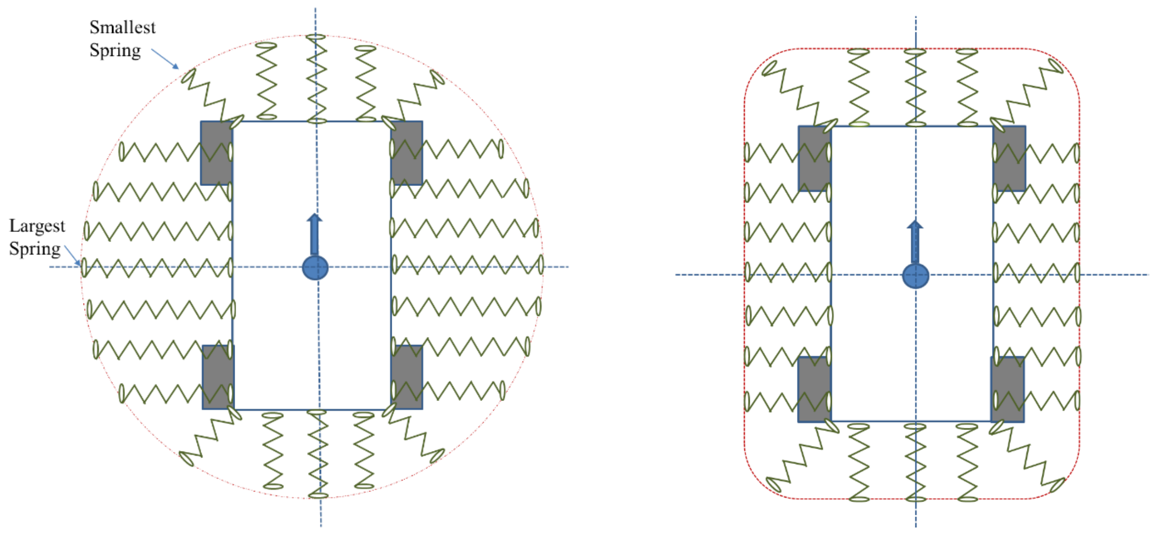


Figure 2.14: Comparison of circular and rectangular force field around robot.

rectangle force field around the robot, as shown in Figure 2.13 (right). As illustrated in the spring analogy of rectangular force field in Figure 2.15(b), all the springs provide uniform force now regardless of their position. This eliminated the spikes in the force-feedback on the joystick.



(a) (b)

Figure 2.15: Spring analogy for (a) circular force field (b) rectangular force field.

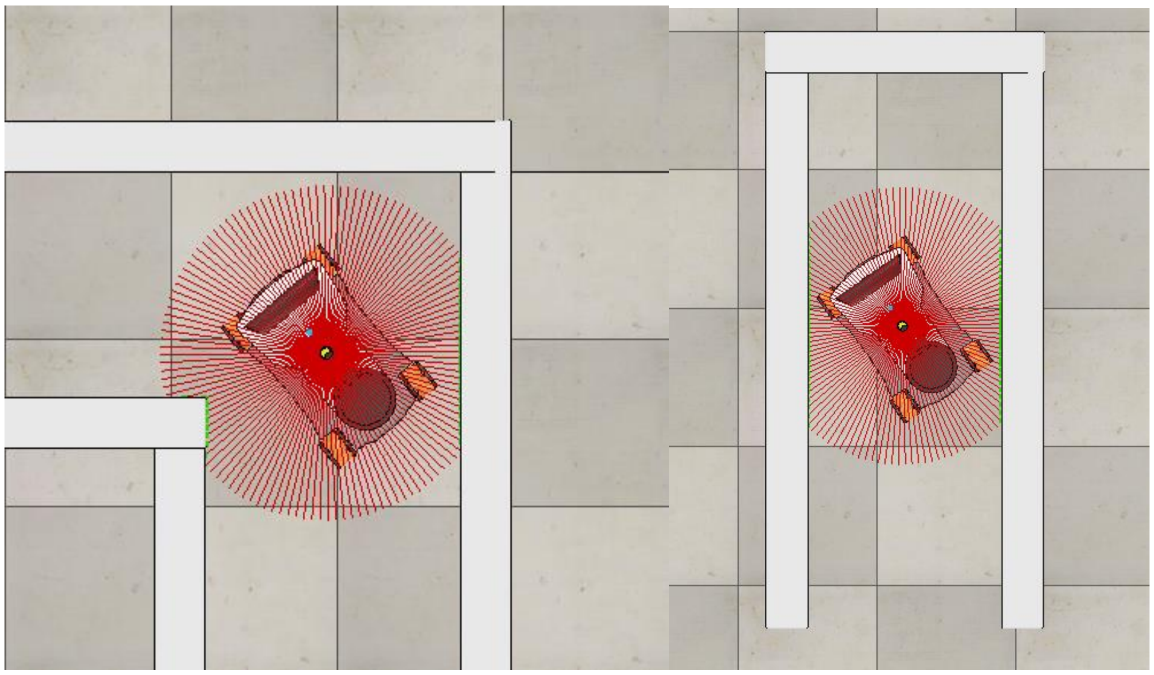


Figure 2.16: Circular force field around corners and confined spaces.

CHAPTER 3

STABILITY ANALYSIS

3.1 Open Loop Transfer Function

In order to determine the stability of the control system given in Figure 2.12 using classical control techniques, it had to be reduced to a single-input single-output system. Since, for these experiments, we are using a simulation instead of a real robot, the closed loop dynamics of the robot can be treated as unity, which simplifies the block diagram. It is assumed that the response in the x and y directions will be symmetric and with gravity compensation. The response of the joystick can be modelled as a second-order linear system. Other nonlinearities must be neglected in order to generate an approximate linear model of the joystick system. The closed loop response of the joystick system to a step input was used to find the transfer function of the joystick and this response was generated only by using a P controller to close the loop. The block diagram in Figure 3.1 illustrates the setup used for step response.

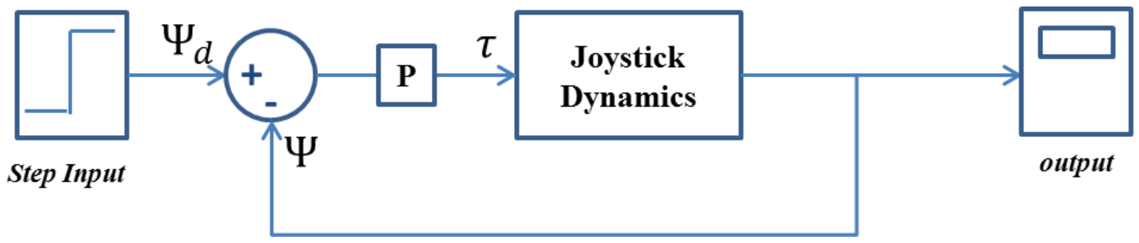


Figure 3.1: Setup for determining joystick response.

This response was used to find the second order characteristic equation that approximates the joystick response:

$$\frac{\psi}{\tau} = \frac{\omega_n^2}{s^2 \quad + 2\zeta\omega_n s + \omega_n^2} \qquad \mathbf{(3.1)}$$

The system is treated as follows:

$$\tau = I\ddot{\psi} + B\dot{\psi} + K\psi$$

$$\frac{\dot{\psi}}{\tau} = \frac{1}{s(IS + B)}$$

$$\frac{\psi}{\tau} = \frac{1}{Is^2 + Bs + K}$$

Substituting the real value of P in equation 3.1 and calculating $\zeta \; and \; \omega_n$ from the step response of joystick, the transfer function of the joystick was found.

The model of the joystick was derived as the following:

$$\frac{\dot{\psi}}{\tau} = \frac{899}{(s + 17.20)s} \qquad \mathbf{(3.2)}$$

In order to validate this model, the response of the transfer function is plotted with the response of the physical system (see Figure 3.2). Even though the physical system is not a perfect match to the derived transfer function, it is sufficient for the root locus stability analysis. Once the model of joystick was derived, the overall block diagram in Figure 2.12 needs to be simplified in order to get an open loop transfer function for root locus stability analysis of the complete system. Figure 3.3 is a simplified version of the block diagram of the complete system, where the robot dynamics is considered as unity because it is being implemented in simulation rather than hardware.

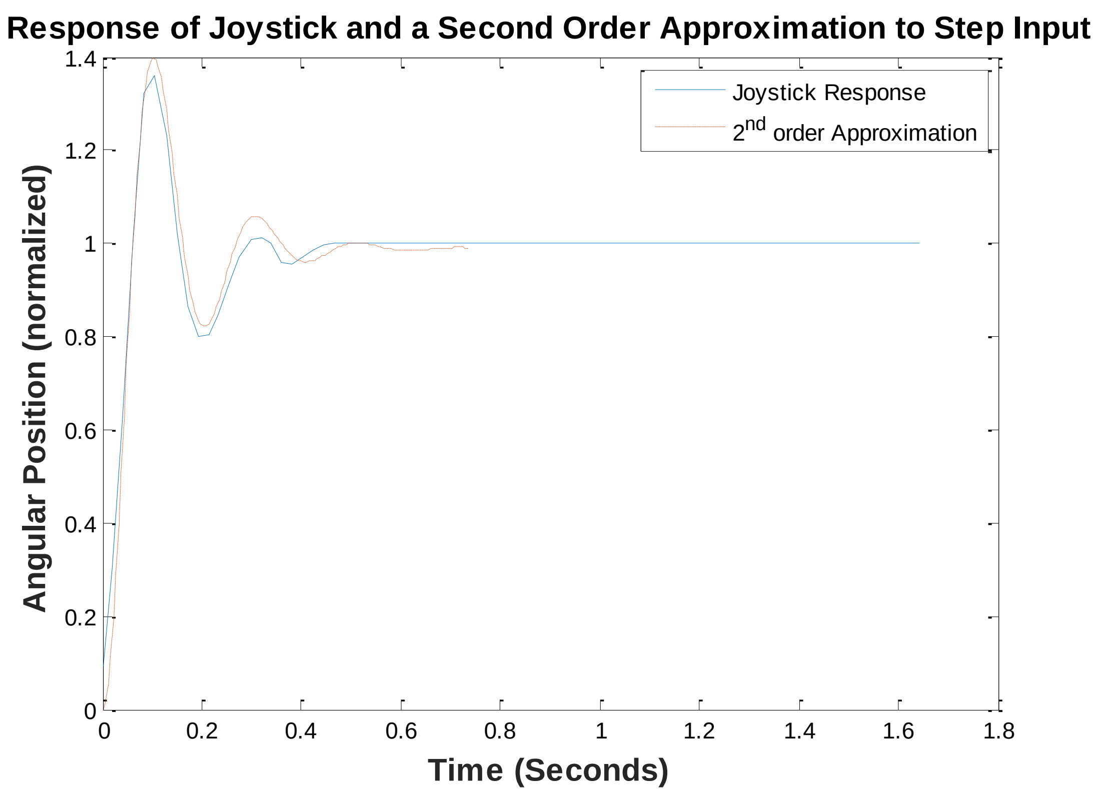


Figure 3.2: Comparison of step response of the physical joystick system to the second order approximation.

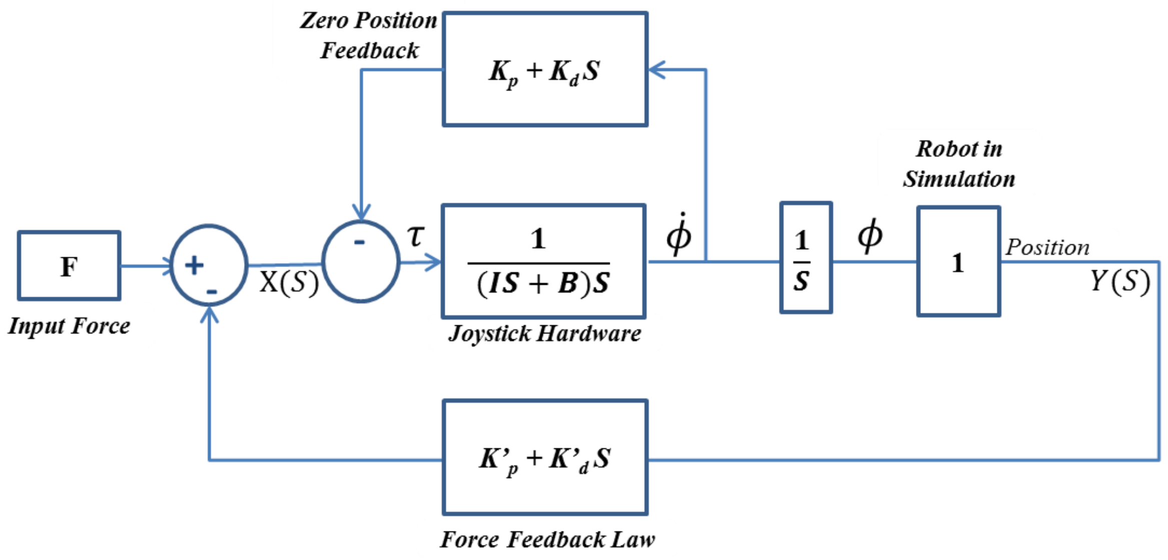


Figure 3.3 Simplified system block diagram.

Here, K_p, K_d are the PD gains for returning the joystick to the zero position, K_p', K_d' are the force-feedback gains, I is the inertia, B is the damping of the joystick hardware, τ is the input torque to joystick, $\dot{\phi}$ is the velocity command for robot, and ϕ is the desired robot position. By using block diagram reduction techniques this is further reduced to the minimal version required for root locus, see Figure 3.4. The open loop transfer function with respect to the feedback law after substituting all the values is given by:

$$\frac{Y(S)}{X(S)} = \frac{101.2\,(s + 6.25)}{s\,(s^2 + 54.95Ss\; + \;629.3)} \tag{3.3}$$

The feedback law adds a zero to the system at K'_p / K'_d = - 6.25. The root locus of the complete system is shown in Figure 3.5. The closed loop poles are marked by a cross in red on the root locus plot. Given the current position of closed loop poles, higher damping ratio (0.8 to 0.9) and faster settling time is expected. If the poles are moved up further, the damping ratio will reduce, which would result in more oscillations on the joystick. It would also result in more overshoot, which is undesirable as that would make joystick oscillate between positive and negative direction and ultimately the robot would also oscillate back and forth. If the poles are moved to right, that would result in slower settling time and if they moved to left, it would result in much faster settling time.

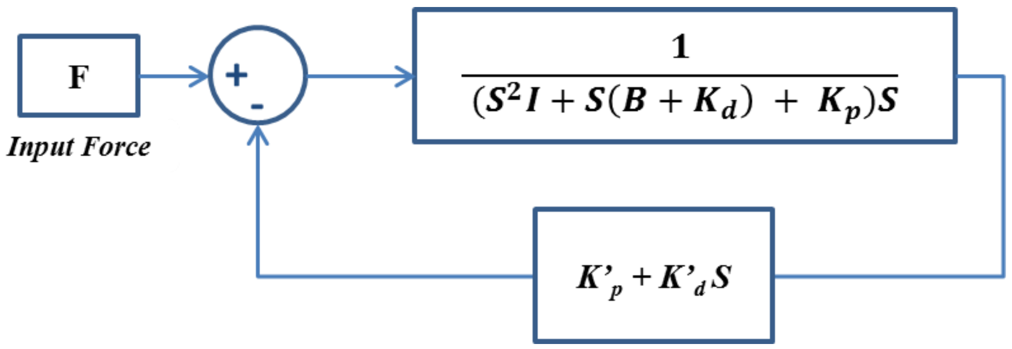


Figure 3.4 Reduced block diagram.

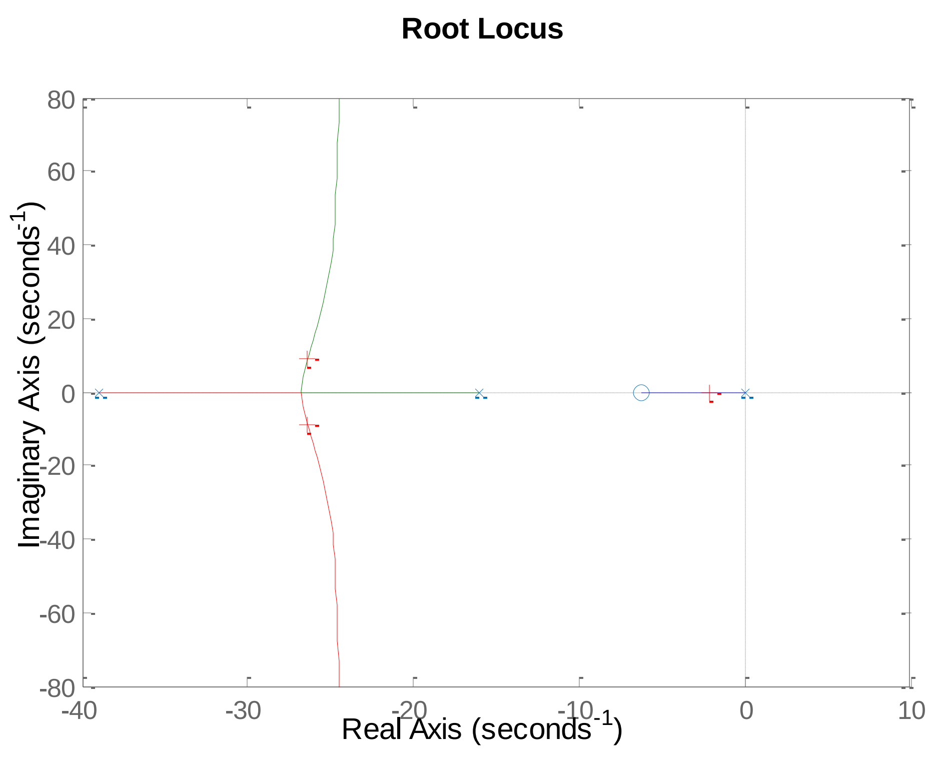


Figure 3.5 Root locus.

CHAPTER 4

HAPTIC FEEDBACK FOR COLLISION AVOIDANCE

4.1 Feedback Objective and Force Field

The key idea is to assist the driver's navigation such that targets are reached and collisions with obstacles are avoided in an intuitive and efficient manner. The joystick should exert 3 degrees of freedom force-feedback on the user's hand in the direction opposite to the obstacle. All 3 degrees are important as the user has to run the robot from a first person view, which is usually the case in real time scenarios like rescue missions where the only view is from a camera mounted on the robot. It is hard to navigate through obstacles without collisions when the user cannot see them in the first person view For example, when taking a turn, the back of the robot might hit a wall. There also might be multiple obstacles through which the robot has to navigate, and only few of them might be visible to the user through the camera.

To improve the quality of feedback in confined spaces, the force field was made rectangular as discussed in Chapter 2. A force field boundary layer was created around the robot, and for initial trials the boundary layer was kept same in the x and y directions. Later after multiple experiments, an optimal boundary layer thickness was tuned and the boundary was kept larger in the x direction and smaller in the y direction to reduce joystick oscillations in the y direction during navigation in narrow spaces (see Figure 4.1).

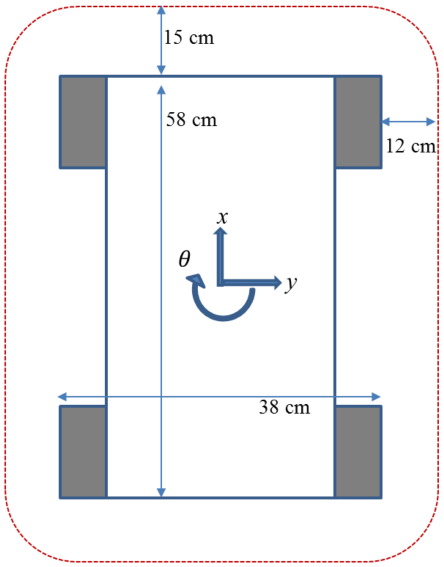


Figure 4.1: Rectangular force field boundary layer.

Various algorithms for implementing force-feedback in 3 DOF were proposed, and their effectiveness was measured using the primitive test scenarios as shown in Figure 4.2. These 3 different scenarios are considered in experiments for their complexity, and are representative of situations where robots have a hard time navigating. In scenario (a) the user has to drive the robot to the end of a tunnel then turn around and return to the start position, while in (b) and (c), the user has to exit at other end of the tunnel.

After driving the robot through these primitives, it was decided to implement the feedback law as a virtual spring-damper system. The virtual damping was introduced to dampen the sudden feedback due to the spring force as it was creating very quick motions in the joystick, resulting in discomfort to the user. The virtual damping encourages slower velocities when approaching obstacles whereas a virtual spring generates feedback so as to maintain a minimum distance between robot and obstacle. Three different force-feedback algorithms are discussed in next subsequent subsections, and these algorithms are tested in the above primitives to select the best one.

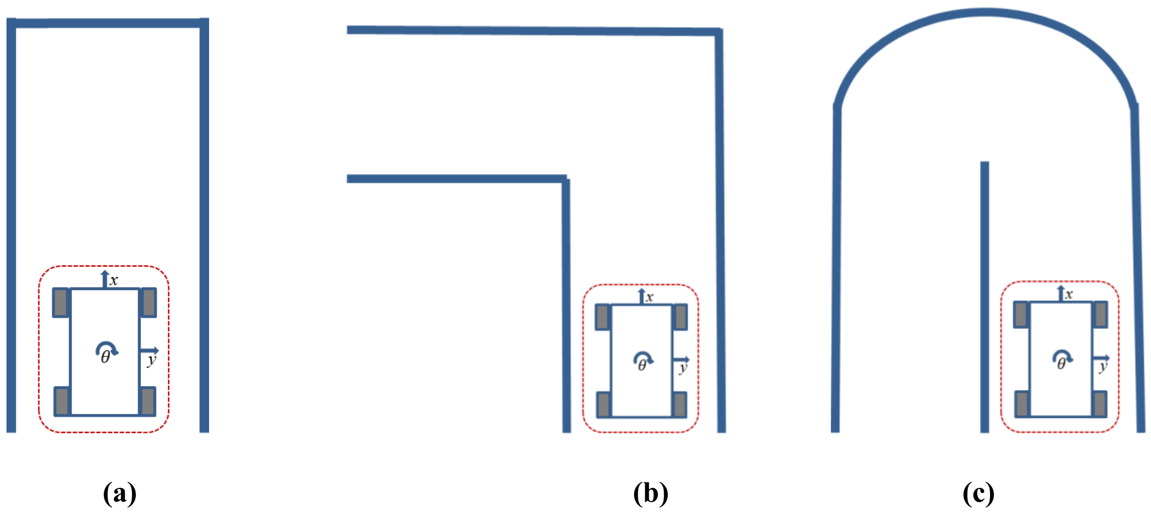


Figure 4.2: Primitives for testing algorithms.

4.2 Force-Feedback as Natural Spring and Damper

This is a basic algorithm representing the natural behavior of springs and dampers, so that when an obstacle comes in the vicinity of the force field, the spring pushes the robot in the opposite direction. Each ray of LIDAR is split into spring's x and y components as shown in Figure 4.3. The force-feedback on joystick was generated according to the following equations:

$$f_{xi} = k_x \, . \, (x_0 - x_i) \tag{4.1}$$

$$f_{yi} = k_y \, . \, (y_0 - y_i) \tag{4.2}$$

$$f_x = b_x \, . \, v_x + \left(\sum_{i=1}^{n} f_{xi} \right) \Big/ n \tag{4.3}$$

$$f_y = b_y \, . \, v_y + \left(\sum_{i=1}^{n} f_{yi} \right) \Big/ n \tag{4.4}$$

$$\tau = b_\tau \, . \, \omega + k_\tau \, . \left(\sum_{i=1}^{n} f_{xi} \, . \, r_{yi} + \sum_{i=1}^{n} f_{yi} \, . \, r_{xi} \right) \Big/ n \tag{4.5}$$

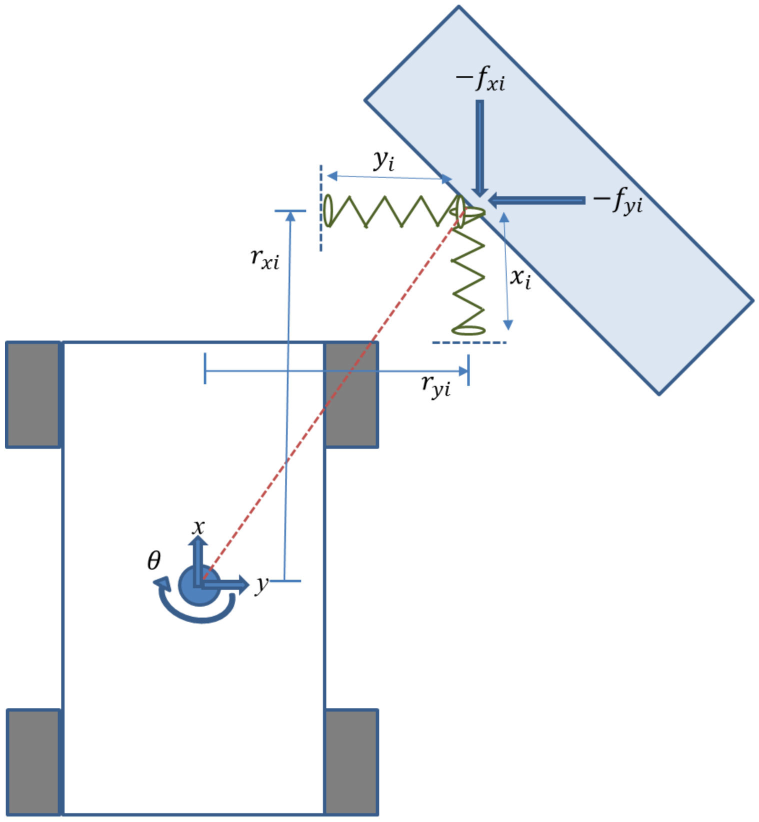


Figure 4.3: Virtual springs in action.

Here, f_x and f_y are the respective cumulative forces in the x and y directions, τ is the effective torque, v_x, v_y, and, ω are the linear and rotational velocities of the omnidirectional robot, b_x, b_y, b_τ, k_x, k_y, k_τ are the virtual damping and spring coefficients in the respective degrees of freedom, x_0 and y_0 are the boundary layer thicknesses, r_{xi} and r_{yi} are the moment arms to the point where each spring is acting, and n is the actual number of LIDAR rays detecting obstacle. The effective rotational feedback is the torque exerted by each spring in the x and y directions multiplied by a gain. The force/torque values f_x, f_y, and τ are fed to the joystick to generate haptic feedback so as to avoid collision of the robot with obstacles, and these feedback values are updated after every LIDAR scan.

The problem with this algorithm is that in corners, an equal and opposite torque is generated from the springs in the x and y directions, such that the net torque doesn't assist the user to take the turn, instead pushing the joystick in the wrong direction, as illustrated in Figure 4.4. Here the x component of the spring applies a torque (f_{xi} **x** r_{yi}) in the clockwise direction, but the spring component in the y direction applies a torque of (f_{yi} **x** r_{xi}) in the counterclockwise direction, counteracting each other. However in most cases f_{xi} is much larger than f_{yi}, forcing the robot to turn clockwise, which is counterproductive to the user, who is trying to rotate the joystick/robot counter clockwise to avoid collisions. This creates confusion and the robot gets stuck in that corner and creates a counter-intuitive user experience. Therefore, some modifications were required in the feedback algorithm so as to make the joystick assist the user in rotating the robot around corners.

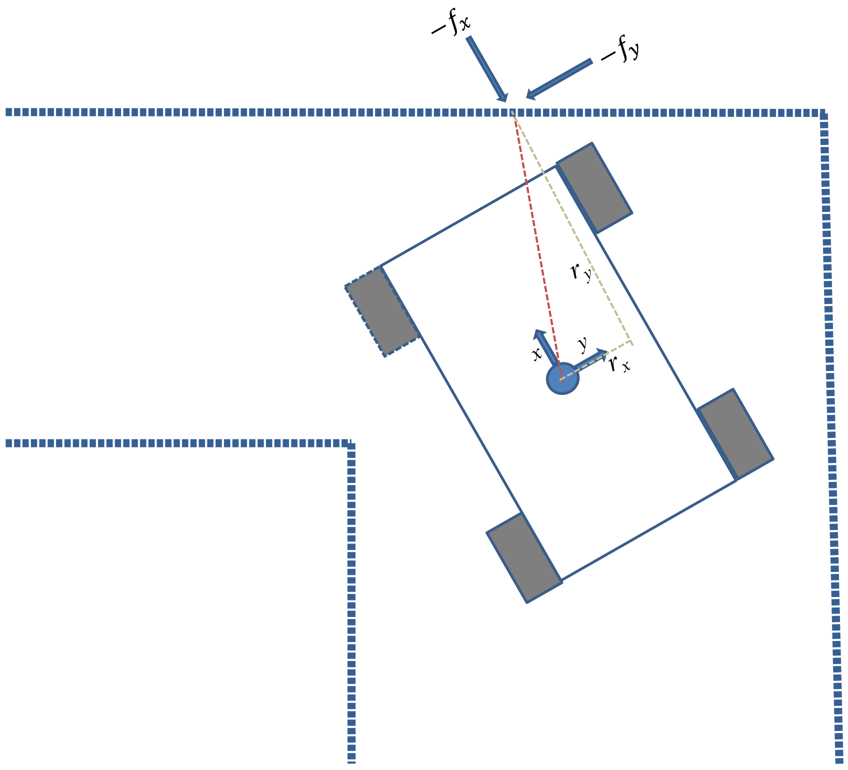


Figure 4.4: Robot stuck in corner scenario.

4.3 Force-Feedback as Quadrant Approach

Due to the limitations of the natural spring and damper approach in corners a new feedback law was introduced, which was named the Quadrant Law. In this approach the robot's force field is divided into Quadrants as shown in Figure 4.5. In this case, F_x and F_y are calculated as before, but the torque is calculated with a different approach. A resultant force was derived from the x and y components of the spring force, and the resultant force was calculated by using the equation 4.6.

$$f_r = \sqrt{\left(\left(\sum_{i=1}^{n} f_{xi}\right)^2 + \left(\sum_{i=1}^{n} f_{yi}\right)^2\right)\Big/ n^2} \qquad (\mathbf{4.6})$$

where f_r is the magnitude of resultant force and k_τ is the torque gain.

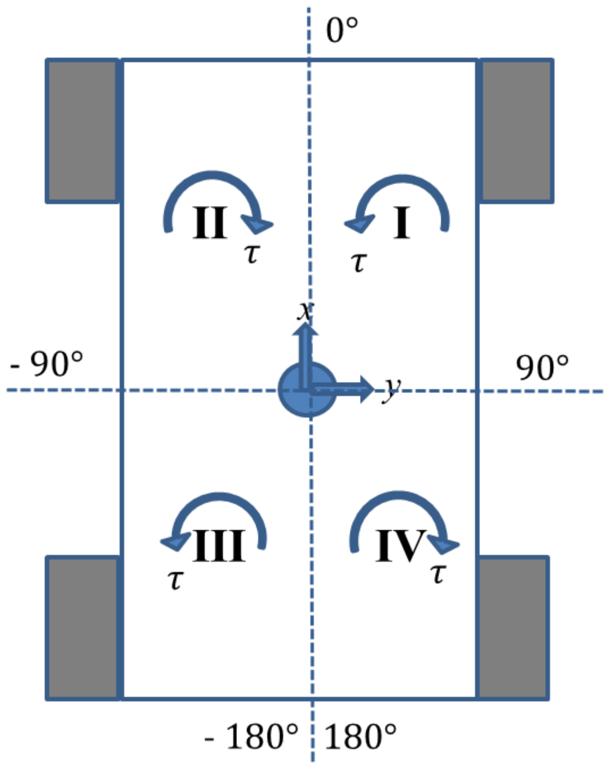


Figure 4.5: Quadrant law.

The torque is calculated using equation 4.7:

$$\tau = b_\tau . \, \omega + (f \textbf{ x } r_a) \qquad \textbf{(4.7)}$$

where r_a is an imaginary moment arm across which the torque is calculated. For experiments, different values of r_a were tried and the best feedback was achieved when it was 10 cm away from center.

The direction of the torque was calculated using the quadrants in which the resultant force acts: if the resultant force is in quadrant I or III, commands are given to generate counter clockwise torque, while if the resultant is in quadrant II or IV, clockwise torque is generated. This approach generated desired trajectories as needed to drive robot out of the corner situations. However, it was realized that since only the direction of the torque was changed when there was a quadrant jump and the magnitude was constant as the moment arm was constant, it was creating a discontinuity in feedback as the feedback abruptly changes direction while maintaining the same magnitude (see Figure 4.6).

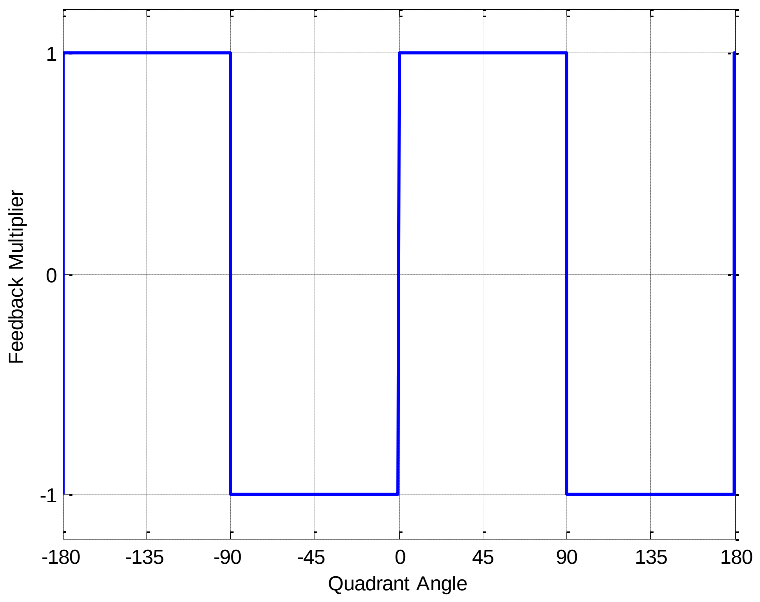


Figure 4.6: Quadrant law, rectangular quadrant transition.

This caused a sudden jerk on the joystick handle that was hard for the user to interpret. In order to get rid of this discontinuity, we implemented a smoothing curve across the theta feedback. This smoothing factor was a sine curve multiplied with the magnitude of the theta feedback, where the direction is determined the same as before on the basis of quadrants. This gave us a smooth transition in between quadrants (see Figure 4.7). This made the magnitude a maximum at the -45°, -135°, 45°, 135° angles, as the smoothing curve was sin(2θ), and a minimum near the quadrant boundaries. This smoothed out the jerkiness in the joystick during the transition between quadrants. However, after multiple tests in the primitives discussed in Figure 4.2, and after plotting the theta feedback data vs. time, it was found that there was still some undesirable behavior occurring in the torque feedback. Although the magnitude was small, the change in direction caused some counterintuitive behavior on the joystick. The plots for this law are discussed in the next subsection for comparison with the new feedback law.

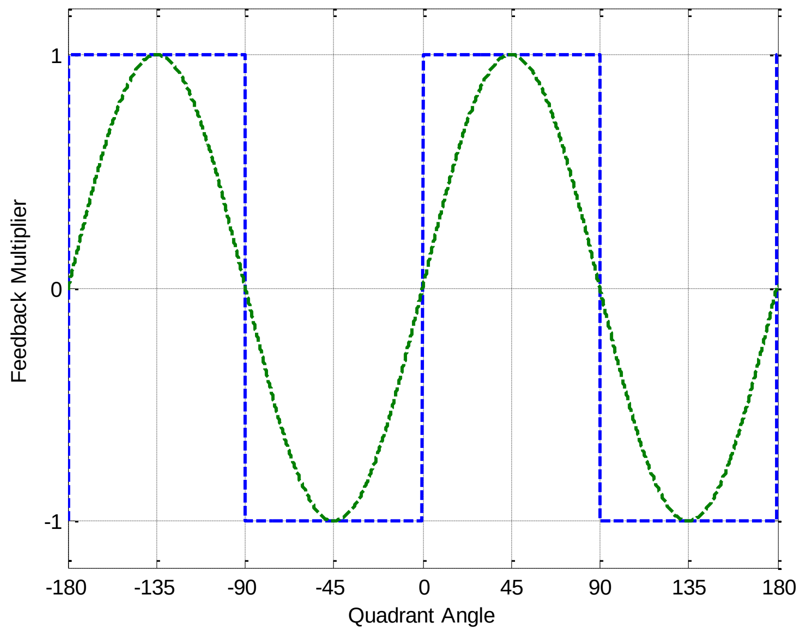


Figure 4.7: Quadrant law, sine smoothening curve.

4.4 Force-Feedback as Component Approach (F_y Law)

This approach is derived from the two laws, the natural spring and damper feedback law, and the quadrant law. In this approach the F_x and F_y forces to the joystick are calculated as in the before two approaches (see equations 4.3 and 4.4), but the torque is calculated in a different manner. In the quadrant approach, it was noted that there was a jerk while there was a transition of quadrants, and in the natural spring and damper approach the robot was being pushed back by the virtual spring in the x direction, so a new law was derived based on what was learned from these two. It was proposed that the torque will only be generated by the virtual springs in the y direction and the torque generated by virtual springs in x direction will be zero (see Figure 4.8).

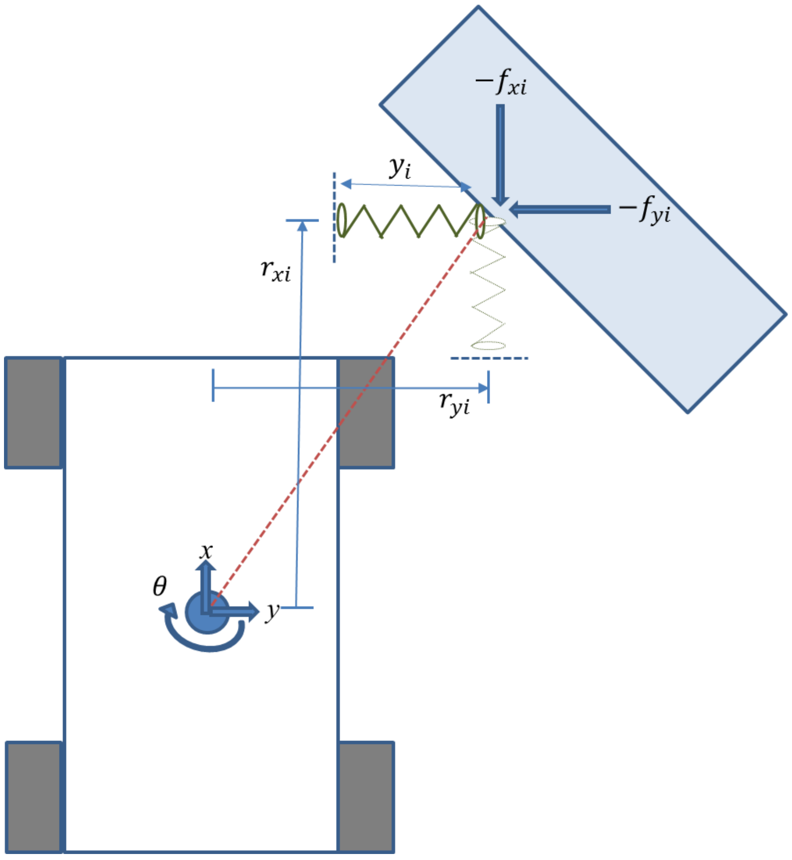


Figure 4.8: Virtual spring forces in F_y law.

Thus, the modified feedback law for rotational torque on joystick is:

$$\tau = b_\tau \, . \, \omega + k_\tau \, . \left(\sum_{i=1}^{n} f_{yi} \, . \, r_{xi} \right) \Big/ n \qquad (\mathbf{4.8})$$

Here, τ is the effective torque, ω is the rotational velocity of the omnidirectional robot, b_τ *and* k_τ are the virtual damping and spring coefficients in the rotation, r_{xi} is the moment arm to the point where each spring is acting, and n is the actual number of LIDAR rays detecting obstacles. The effective rotational feedback is the torque exerted by the spring's y direction multiplied by a gain. The f_x, f_y, and τ values are fed to the joystick to generate haptic feedback so as to avoid collisions of the robot with obstacles, and these feedback values are updated after every LIDAR scan. This approach and the corresponding feedback law was named the F_y Law, as only F_y forces are responsible for generating rotational moments.

This algorithm was also tested on the primitives in the Figure 4.2. and it was observed that the F_y law did not make unnecessary transitions in between quadrants and has a smooth feedback law without the need of any smoothing curve. This can be observed in Figure 4.9 at around 7.5 seconds where the quadrant laws make a transition in positive feedback which results in a momentarily clockwise feedback on the joystick while the F_y law never makes any transition to positive feedback.

4.5 Experiments

For testing the functionality of the feedback law given in section 4.4 (F_y law), a complete three-dimensional scenario was created in V-REP to run the omnidirectional robot (see Figure 4.10). This simulation was based on the primitives (Figure 4.2) used

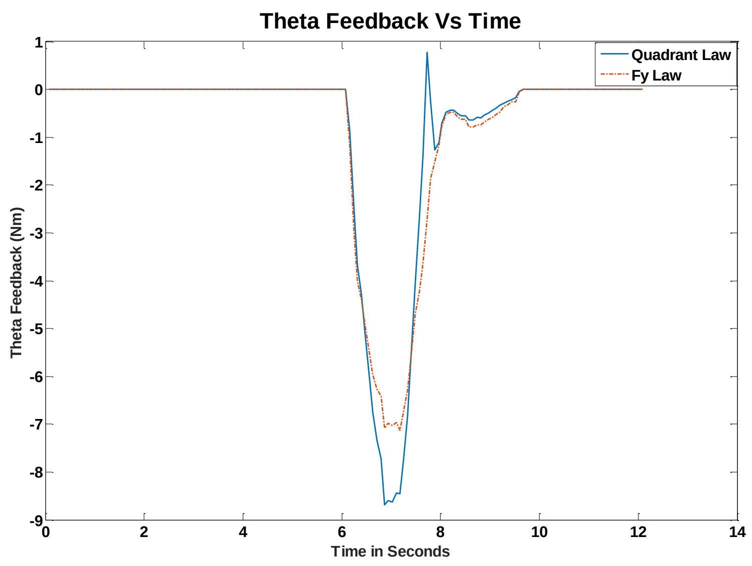


Figure 4.9: Comparison of theta feedback in quadrant law and F_y law.

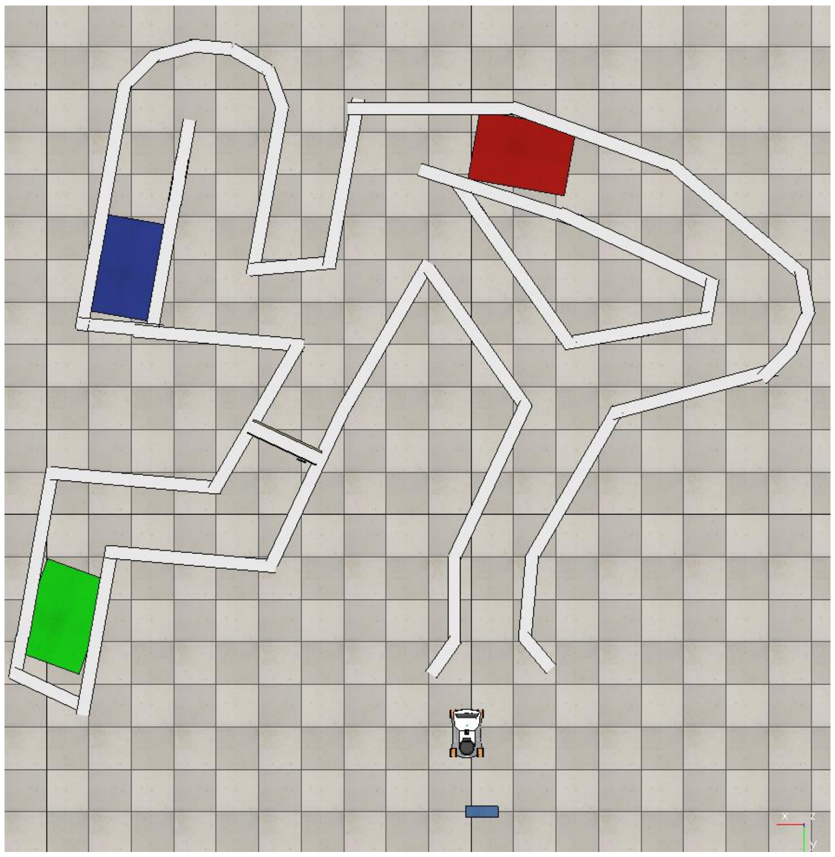


Figure 4.10: Complete scenario to test the feedback law.

earlier for testing effectiveness of different algorithms. The task for the operator was to drive the robot using the haptic joystick through all the colored blocks and then return back to the starting position and touch the small blue block. In each trial, the number of collisions, the total time to complete the maze, the x, y, and torque feedback vs. time, and the x, y, and theta trajectory vs. time were recorded. Two different sets of human trials were done to prove the hypothesis of this research as per University of Utah Institutional Review Board (IRB) regulations, IRB Number: 00064011.

The first set of trials was done by 8 human subjects using the quadrant feedback law. However, later, it was realized that a better law could be designed in order to eliminate the quadrant switching, and therefore the data from those tests were ultimately disregarded. However, some of that data will be analyzed in the next section. The second set of human trials was done again by 8 different human subjects using the F_y feedback law, and the tests were repeated for each subject with different parameters. Since there were many parameters that could have been varied for trials, the ones that were thought to be most worth experimenting were selected. Also the total experiment time for each user was limited to a maximum of 1 hour, and the number of runs was limited to 10 per subject. We tested the effectiveness of the F_y force-feedback law by comparing it with no force-feedback, by changing the compliance center of robot, and also with zero rotational feedback. Four different compliance centers were chosen so as to see its effect on the quality of feedback as observed in the traditional peg in hole problem (see Figure 4.11).

The first configuration was with the compliance center at the center of robot, the second with the compliance center moved 10 cm towards front of the robot, the third was 20 cm away, and, in the fourth configuration, it was moved 10 cm towards the back of robot. The 10 trials were ordered as follows:

- 3 with no force-feedback
- 3 with force-feedback, compliance center at center of robot
- 1 with force-feedback, compliance center at +10 cm
- 1 with force-feedback, compliance center at +20 cm
- 1 with force-feedback, compliance center at -10 cm
- 1 with x and y force-feedback but zero torque feedback

For each subject the order of first 6 trials were changed, i.e, some subjects started with force-feedback on the joystick and some started with the no force-feedback, but the order of last 4 was same, as we already tested the effectiveness of these in the primitives designed before. The user had to take a different route to complete each round; and there were colors on the path to mark different routes. Three different combinations of routes were chosen: Red-Green-Blue, Red-Blue-Green, and Blue-Green-Red to eliminate the subject's learning curve of the path. The above sequences were chosen as the path length is same in all these 3 routes, and therefore, a good comparison of time to completion could be made. These 3 route sequences were also shuffled for every user so as to generate complete randomness in the trials. A closed door was also put in path in the scenario (see Figure 4.12) so as show the importance of force-feedback over autonomous robots, as an autonomous robot would consider a door as a wall and turn back, but if a user is driving they can break open the door by overriding the haptic feedback on the joystick.

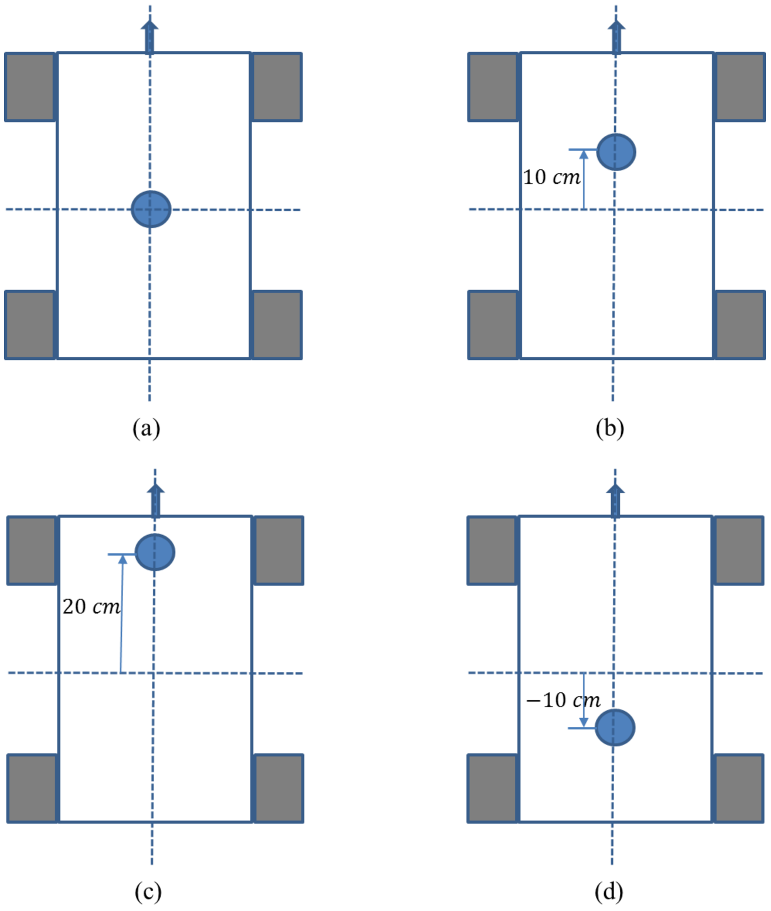


Figure 4.11: Different compliance center configurations.

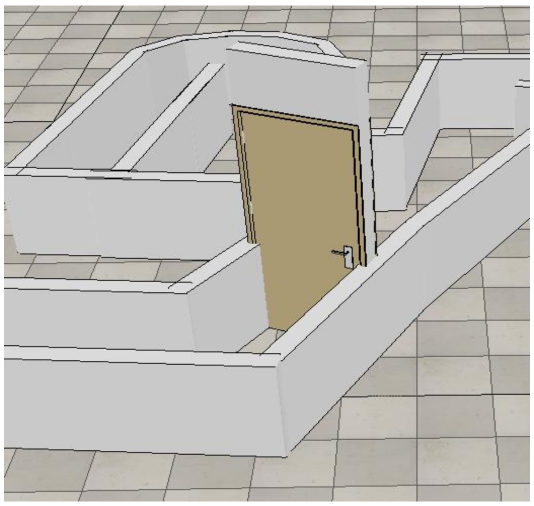


Figure 4.12: A door in the path of robot.

CHAPTER 5

RESULTS AND ANALYSIS

5.1 Number of Collisions and Time to Complete

The number of collisions and time to complete the run was recorded manually on the paper and was analyzed later after all the trials were done. The bar plot in Figure 5.1 shows the average number of collisions for all test subjects, T_1 to T_3 are the tests with force-feedback, T_4 to T_6 are the tests without force-feedback, in T_7 the compliance center is moved 10 cm towards front of robot, T_8 has a compliance center 20 cm away, and the T_9 has a compliance center 10 cm towards back of robot. T_{10} trials were done by turning off the rotational feedback, such that the user observes only the X and Y feedback. T_{avg} is the average of all the trials while the robot was running with force-feedback and without force-feedback for all subjects.

From the bar plot it is evident that the total average number of collisions without force-feedback are as high as 34 collisions while with force-feedback they are reduced to 5. It can also be observed that from T_1 to T_3 as well as from T_4 to $T_{6,}$ there is a strong learning curve, and as the user learns how to drive the omnidirectional robot using the joystick, the number of collisions decreases substantially. Also, for most of the users the number of collisions with force-feedback converges to zero with as low as in 2 trials, but without force-feedback, this is still high at around 25 even after 3 trials.

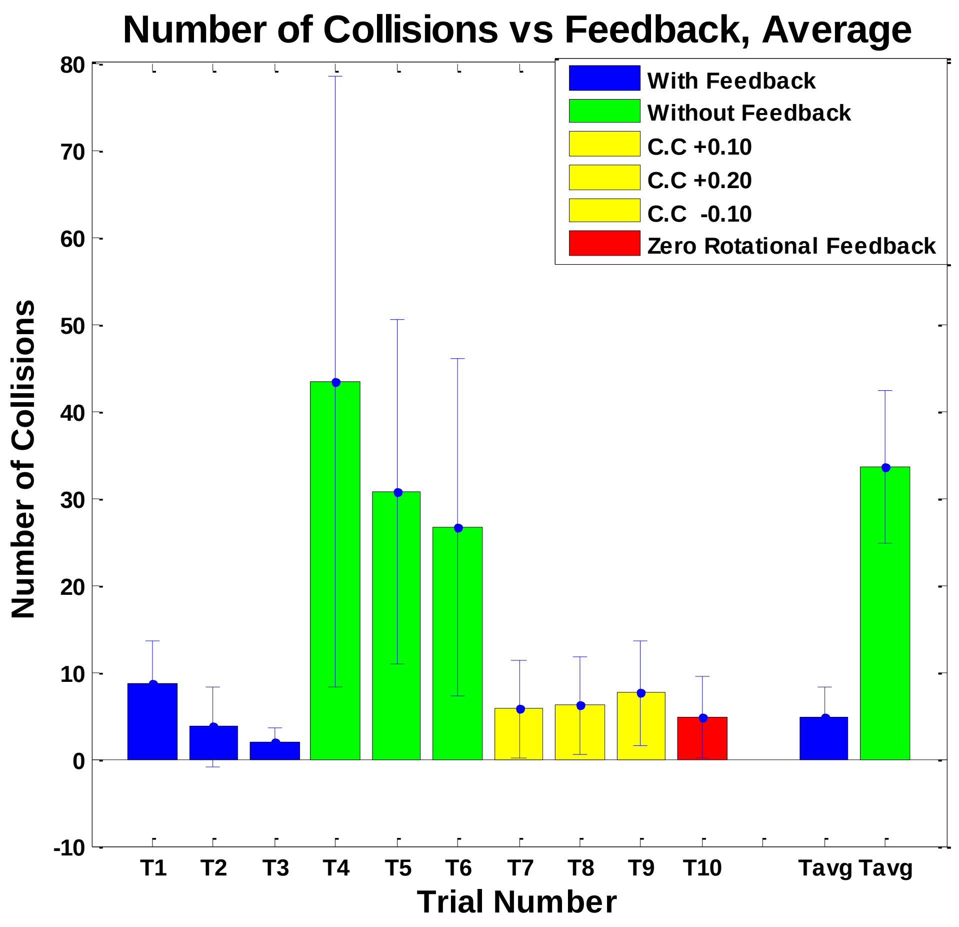


Figure 5.1: Bar plot for average number of collisions with standard deviation.

The number of collisions appears with force-feedback appears to remain small even when changing the other parameters like compliance center and rotational feedback; therefore, these parameters are explored more in the next subsection by comparing them in terms of smoothness in trajectories and feedback.

Comparisons were also made in terms of time to complete the run, but surprisingly there is no significant difference in between them, see Figure 5.2. The average time to completion with force-feedback is little less than the time in other categories, but it is not a significant difference. This difference can be attributed to more collision without force-feedback and the user trying to control robot after collision.

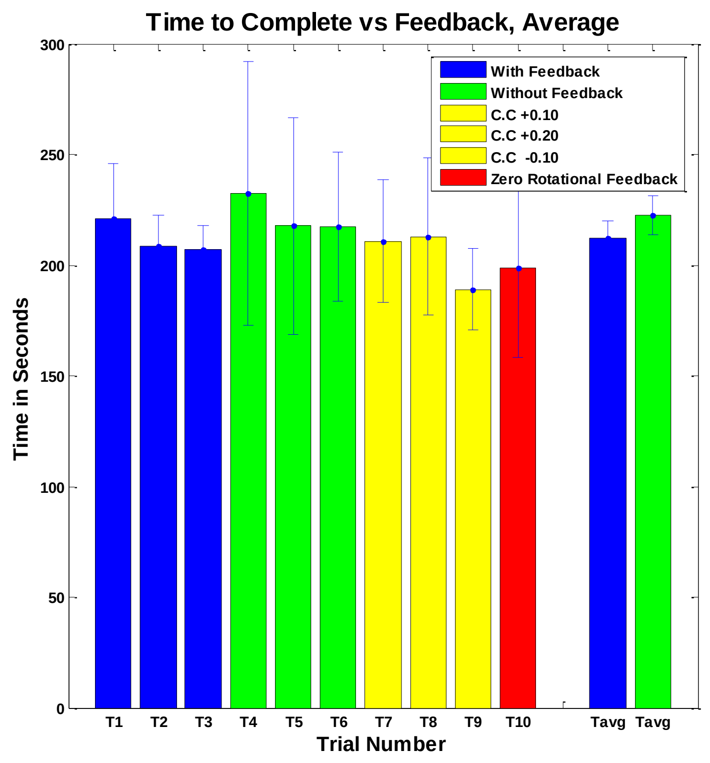


Figure 5.2: Bar plot for average time to completion with standard deviation.

We also analyzed the best and worst user data, from the pool of various participants, see plots in Figure 5.3. This is the best user data and the maximum number of collisions for this participant was 1 with haptic feedback as compared to 19 without haptic feedback. The haptic feedback data converge at zero in the second trial, but the one without haptic feedback does not seem to converge and has 18 collisions towards the end, thus putting the average at 17 collisions. Figure 5.4 below depicts the worst user data from the pool of participants, here, the maximum number if collisions starts at 11 with haptic feedback and eventually narrows down to 4 collisions, but without haptic feedback the number of collisions is rather high, starting from 124 collisions and narrowing down to around 60 collisions. On average this user made approximately 7 collisions with haptic feedback and 83 colisions without haptic feedback.

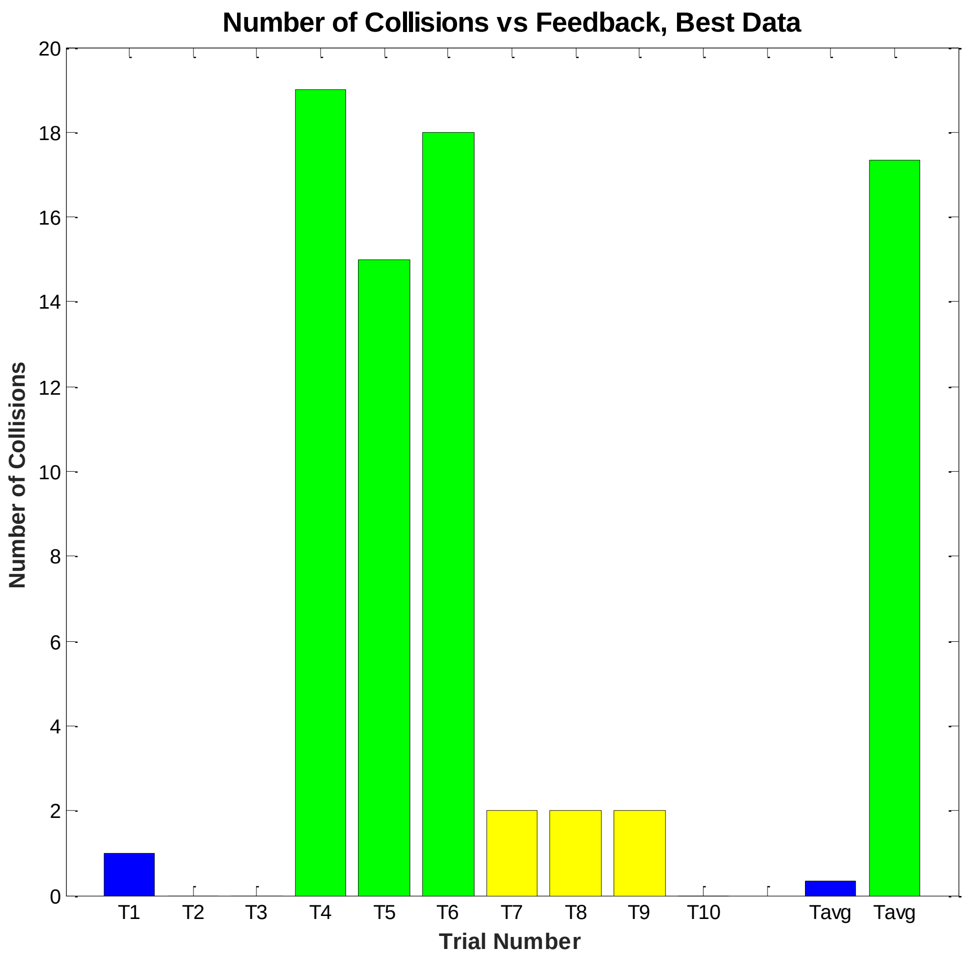


Figure 5.3: Bar plot for number of collisions, for best subject**.**

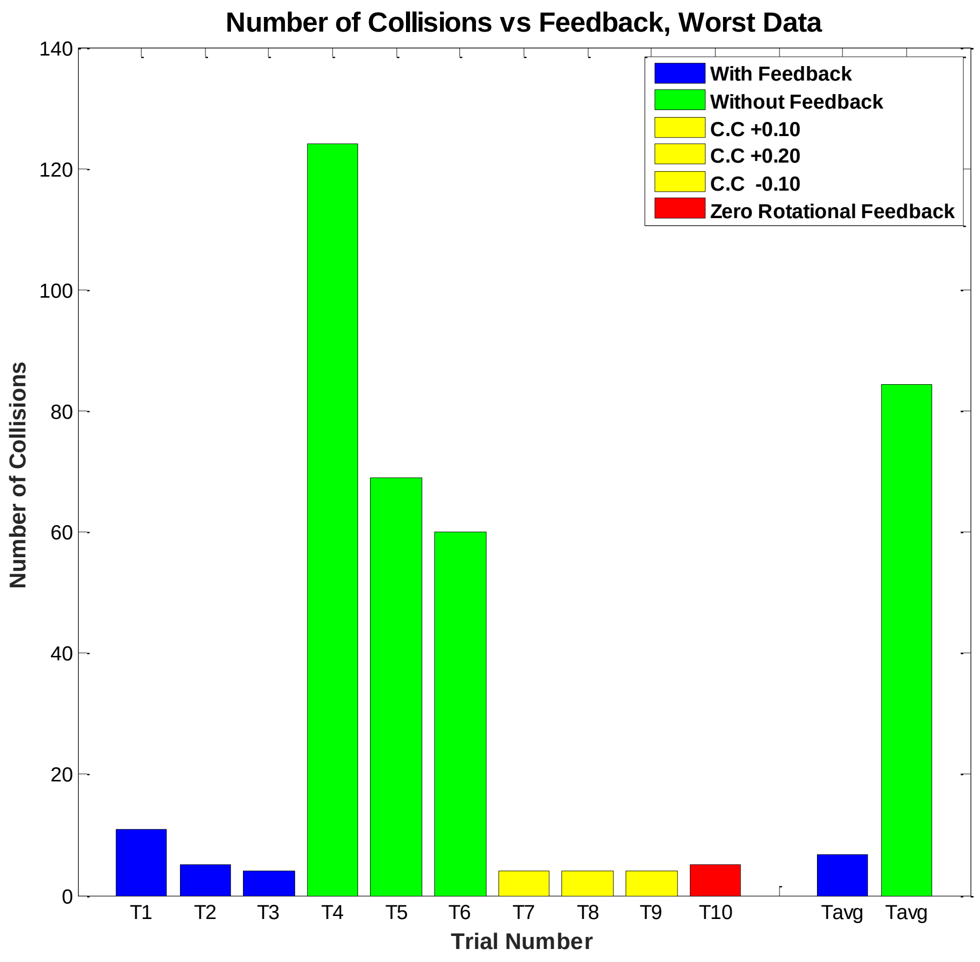


Figure 5.4: Bar plot for number of collisions, for worst subject**.**

5.2 Feedback and Trajectory

By this point it is evident from the data that the use of haptic feedback on the joystick for obstacle avoidance reduces the number of collisions. In order to prove statistical significance an ANOVA was done, and it could be said by 99% confidence that the number of collisions is reduced significant when compared to no haptic feedback. In this subsection, we will extend our discussion so as to see which parameter performs better in terms of providing a smooth feedback and trajectory. The trajectory in x, y, and theta with haptic feedback is compared with the respective trajectories when the compliance center is moved to different spots as well as with trajectories where there is no rotation feedback. Similar comparisons were made for the haptic feedback in the x, y, and theta directions.

These comparisons were made from the dataset recorded from the various user trials; since the whole scenario is big we decided to use a partial dataset based on the primitives discussed before (see Figure 5.5). Please note that the subscenario with the blue block in it corresponds to the primitive in Figure 4.2 (a) and (c) merged together and the dataset obtained from it is called the blue dataset, while the subscenario with the green block corresponds to the merged primitives from Figure 4.2 (a) and (b) and the subsequent dataset is called the green dataset. The blue and green datasets for all the trials were compared against the dataset from the trials with haptic feedback and the compliance center at the center of robot. The plot in Figure 5.6 is a rotational feedback comparison plot between various compliance center configurations proposed in Figure 4.11 vs. when the compliance center is at the center of robot. This is based on the green dataset, but similar observations were made for the blue datasets too. Visually it was noticed in a qualitative sense that the torque feedback was smoother when the compliance center was at the center

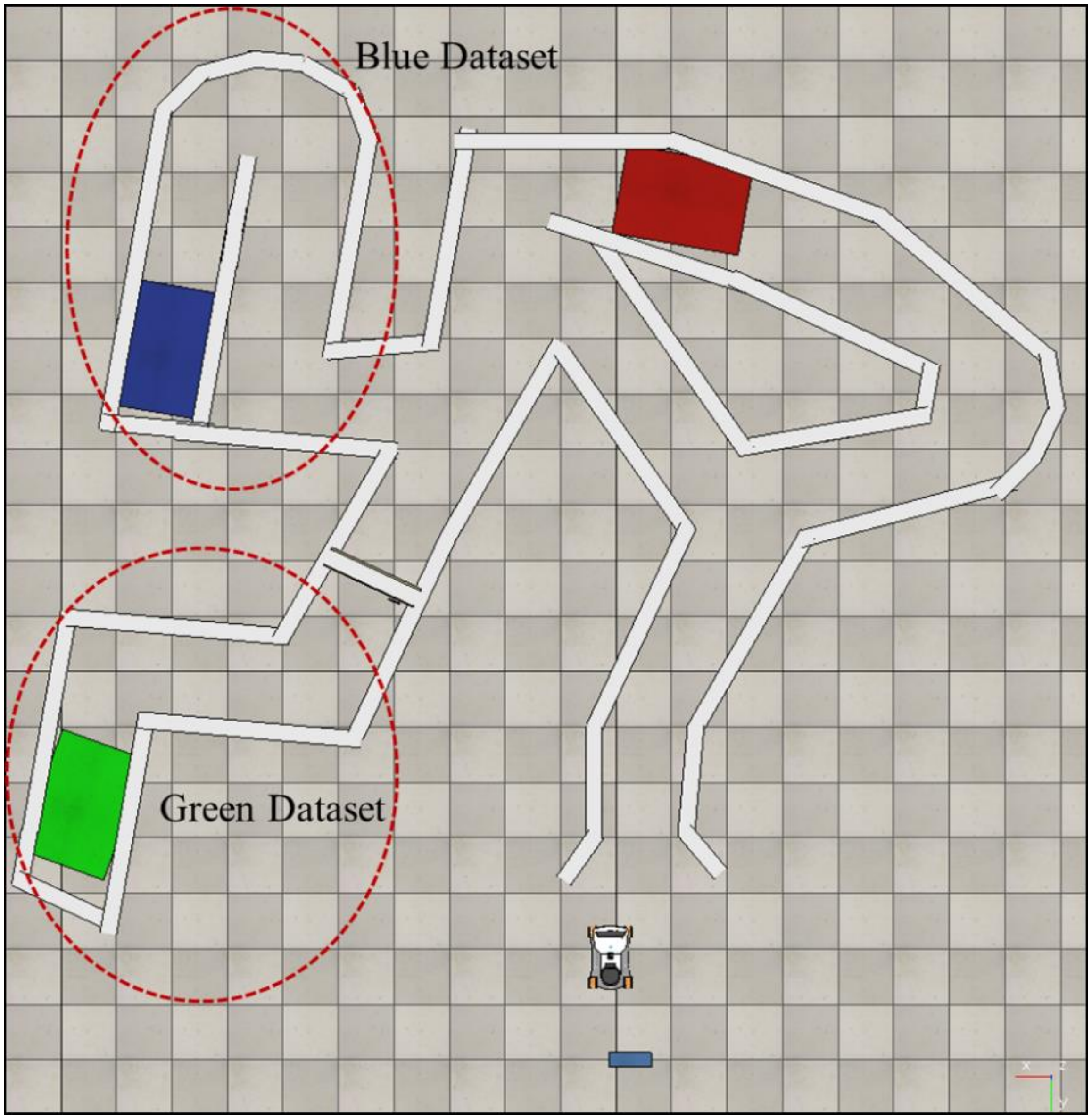


Figure 5.5: Datasets for analysis.

of robot as when compared to the other compliance center positions, and similar trends were noticed for the feedback in x and y directions. The trajectory was also smoother in all 3 DOF when the compliance center was located at the center of the robot, as shown in Figures 5.7, Figure 5.8, and Figure 5.9. From these plots it can be said that the trajectory was smoother when compared to the other compliance center configurations under observation.

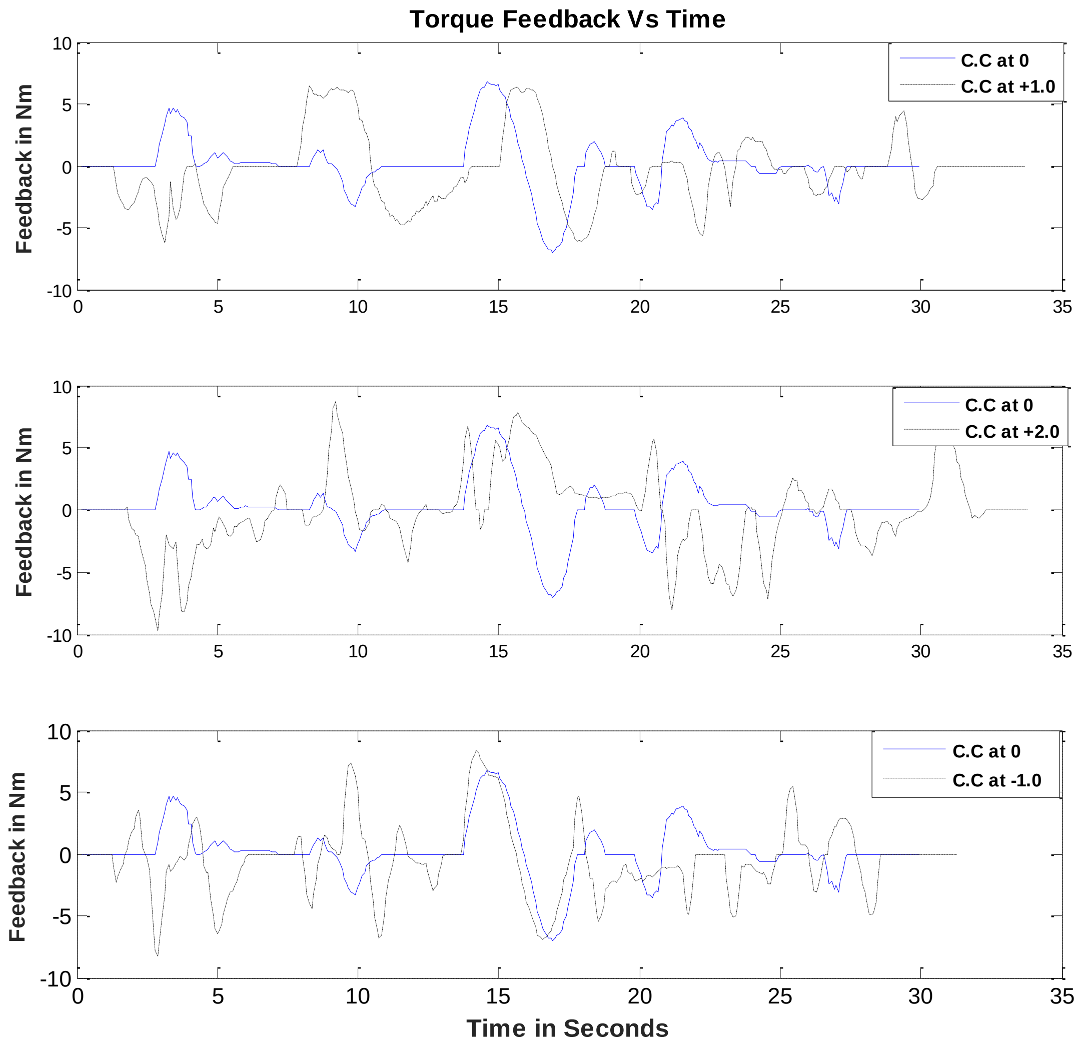


Figure 5.6: Torque feedback vs. time plot for various compliance center configurations.

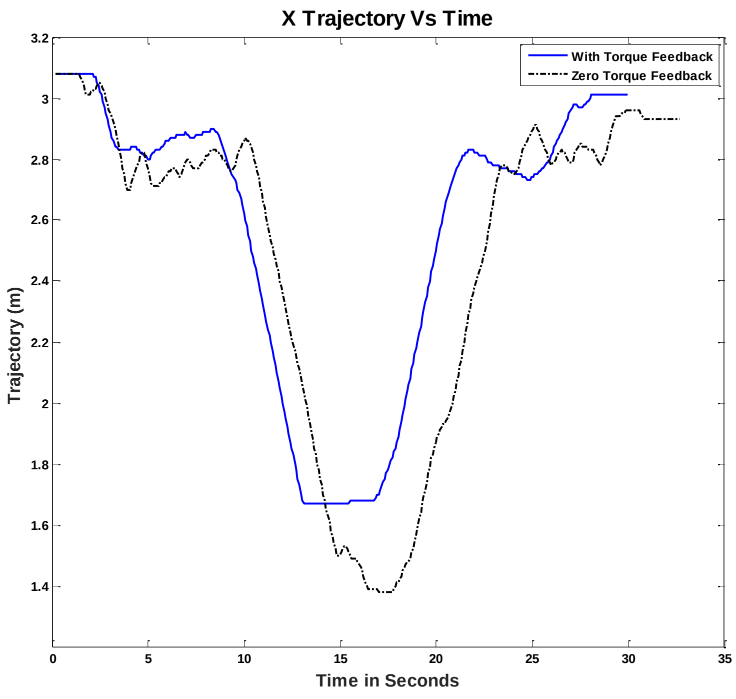


Figure 5.7: Sample x trajectory vs. time plot with and without torque feedback.

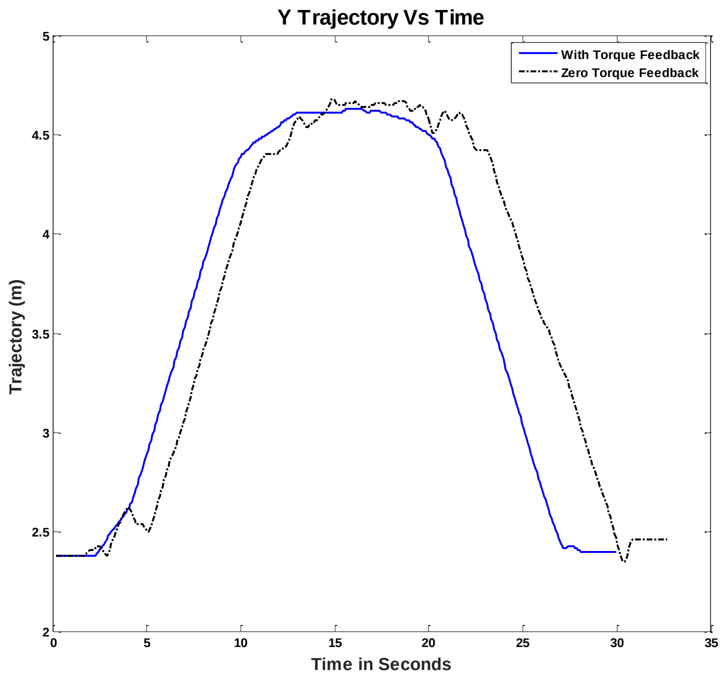


Figure 5.8: Sample y trajectory vs. time plot with and without torque feedback.

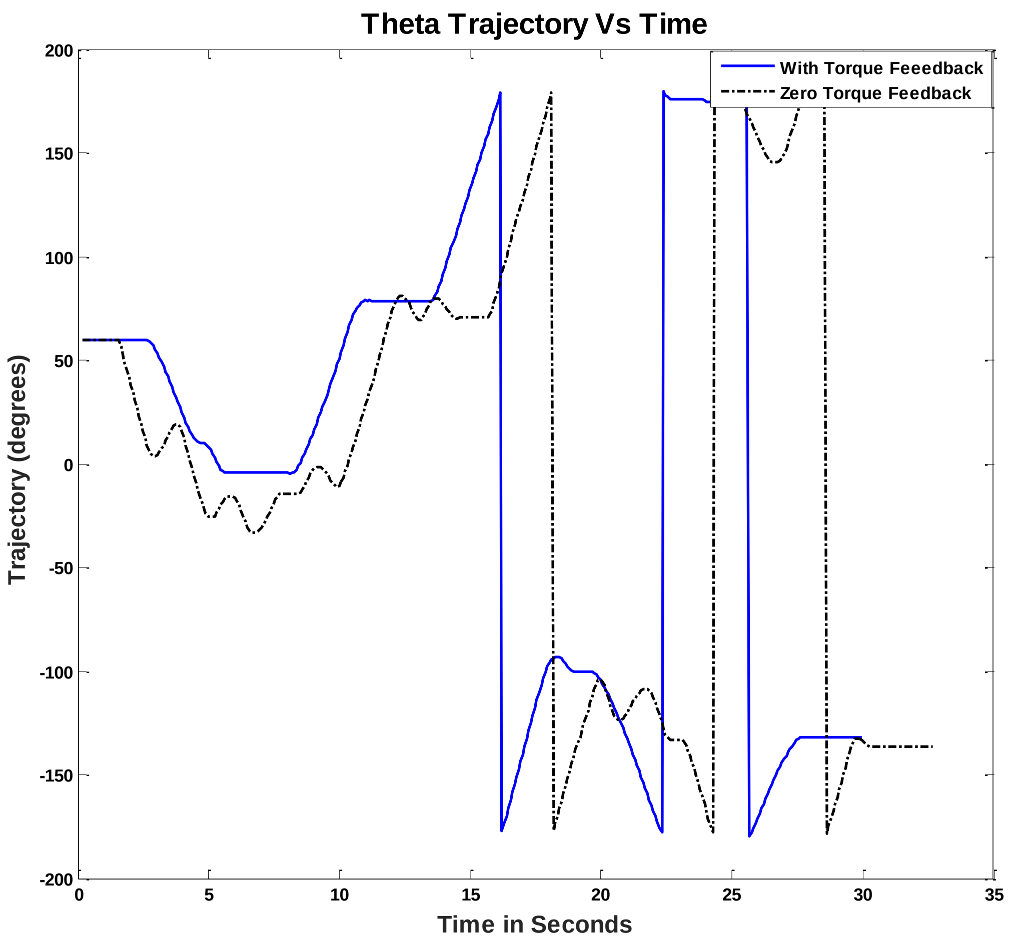


Figure 5.9: Sample theta trajectory vs. time plot with and without torque feedback.

In order to draw conclusions about the results in general, a quantitative metric was derived for determining the smoothness of the trajectory and the the haptic feedback. First, a Fast Fourier Fransform (FFT) of all the individual datasets of the primitives in the scenario was done (a FFT of Y-trajectory is shown in Figure 5.10). The smoothness of the data can then be observed by looking at the frequency content in the FFT.

The DC gain/low frequencies in the FFT represent the intended motion of the robot, while the unintended/unwanted oscillations that were qualitatively observed appeared to show up as quantifiable noise in the 0.8 Hz to 2 Hz range of the FFT (see Figure 5.11).

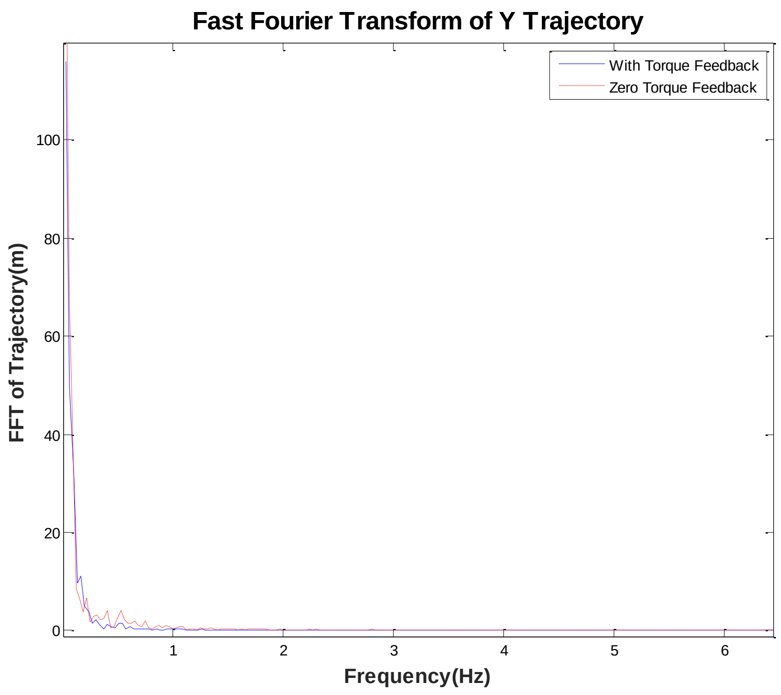


Figure 5.10: Fast Fourier transform of y trajectory.

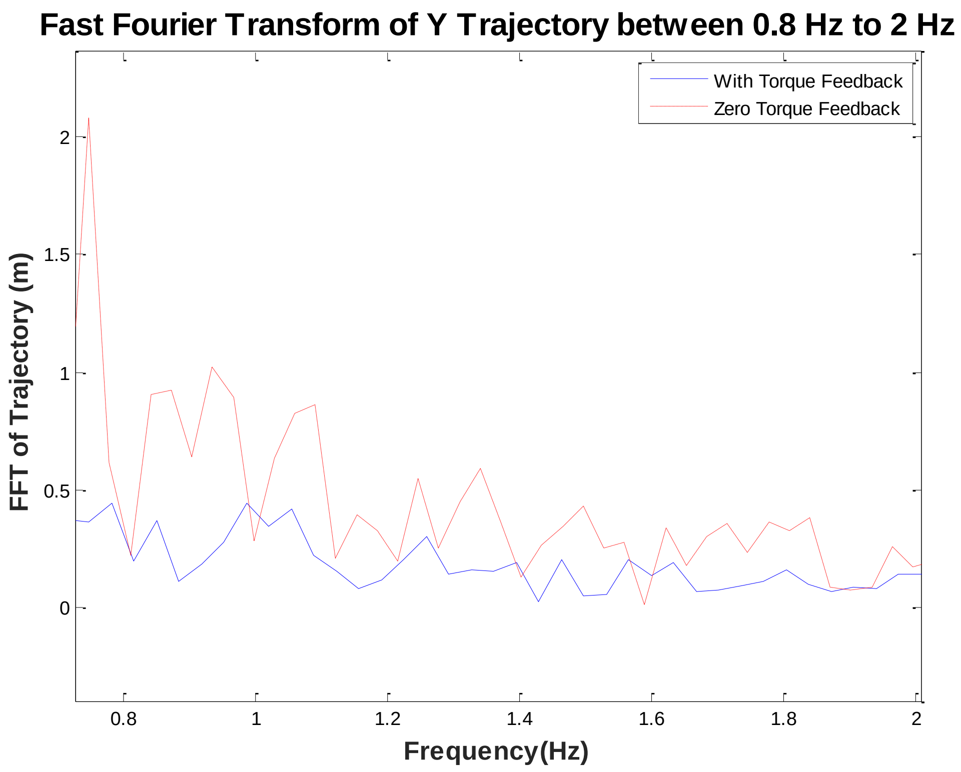


Figure 5.11: Fast Fourier transform of y trajectory in frequency range 0.8 to 2Hz.

By looking at the picture it was evident that there was more noise in the trajectory when the rotational feedback was turned off as compared to when the rotational feedback was on. The power of these signals was found in order to generate a metric for noise comparison. This power of noise was calculated using the sum of the squares of the absolute value of the FFT curve in the frequency range 0.8 to 2 Hz (see Figure 5.12). For a fair comparison between the trajectory and feedback data between different trials by a user, these data were normalized by dividing the power in the 0.8 to 2 Hz range by the power in the entire frequency range.

$$Power = \sum abs(fft_{Trajectory})^2 \qquad (\mathbf{5.1})$$

This power of noise in the trajectory or feedback is inversely proportional to the smoothness of the data, i.e, the smaller the power value the more smooth the trajectory is and vice versa.

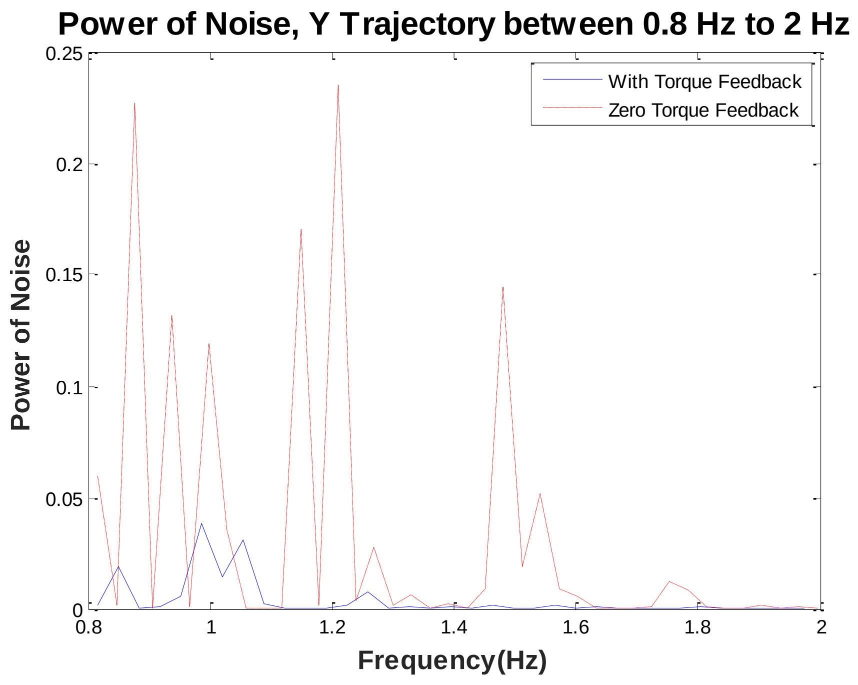


Figure 5.12: Power of noise in y trajectory between frequencies 0.8 to 2Hz.

After finding all the power of the signal by fast Fourier transform, an Anova of variance that produces a “u” value for the feedback as well as trajectory was used. Only “u” values less than 37 (95% confidence) could be used to claim a statistically significant effect on the means of the data, according to the U test. First the “u” values for the force-feedback data were analysed, comparing the algorithm when the compliance center was at center of robot to when the compliance center is moved to different spots, as well as the case when there is no rotational feedback. All of the “u” values were greater than 37, meaning that changing the compliance center has no statistically significant effect on the smoothness of the force-feedback data in all three degrees of freedom. Then we analyzed the “u” value for the trajectory data in the theta direction (Table 5.1) comparing the different positions of the compliance center, and it was found out that there is no statistical significance when the compliance center is moved to either +20 cm or -10 cm or +10 cm.

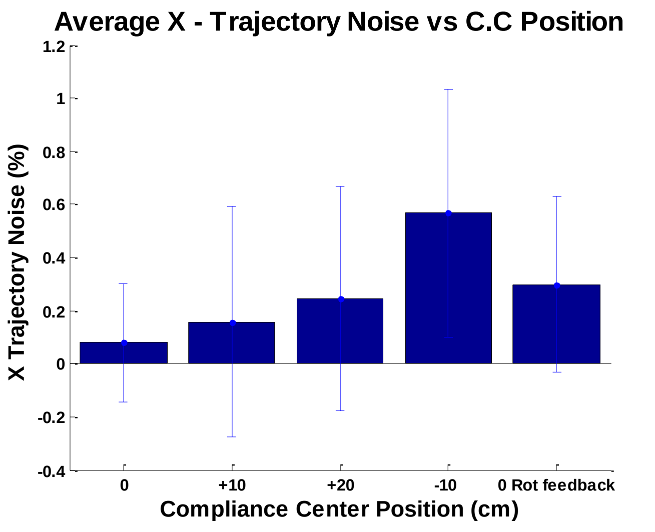
Table 5.1: Different ‘u’ values and their statistical significance.

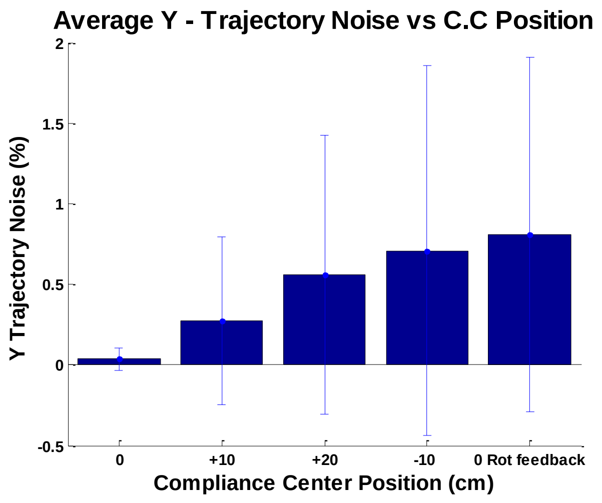
	Data 1	**Data 2**	**'u' value**	**Statistical Significance**
Theta Trajectory	C.C at 0	C.C at +10 cm	50	Not Significant
		C.C at +20 cm	65	Not Significant
		C.C at -10 cm	59	Not Significant
		zero torque feedback	63	Not Significant
X Trajectory	C.C at 0	C.C at +10 cm	51.5	Not Significant
		C.C at +20 cm	31	Significant
		C.C at -10 cm	8	Significant
		zero torque feedback	12	Significant
Y Trajectory	C.C at 0	C.C at +10 cm	38	Significant
		C.C at +20 cm	40	Significant
		C.C at -10 cm	19	Significant
		zero torque feedback	24	Significant

This means that there is no statistical change in the smoothness of the theta trajectory when the compliance center is at any other place than center or when there was no rotational feedback. The “u” values for X trajectory were also calculated and it was noticed that there was some statistical significance when the compliance center was moved at -10 cm and +20 cm, as well as when there was no rotational feedback but not much when the compliance center was moved to +10 cm. So the X trajectory becomes noisier when the compliance center is moved to -10 cm or +20 cm and there is no statistically significant change in the +10cm compliance center positions.

The Y trajectory was also analyzed and it was interpreted from “u” values that if there is no rotational feedback on the joystick, the Y trajectory is statistically worse when compared to the trajectory with rotational feedback. The trajectory was also worse when the compliance center was moved to +20 cm, +10 cm, and -10 cm. The average and standard deviation of the noise in trajectories when the compliance center was moved to different spots and with no rotational feedback is shown in Figure 5.14.

From all the analysis above it is concluded that the trajectory is better when the compliance center of the robot is at center of robot. With no rotational feedback there is statistically no alteration in the theta and x trajectory but a statistically significant degradation in y-trajectory with a confidence interval of 95%. Hence, the best trajectories in all 3-DOF combined are generated by the 3-DOF haptic feedback with the compliance center at the center of robot. It should be noted that these results are not similar to the peg in hole problem as in the typical peg in hole problems it works best when the compliance center is at front of robot. This points to some interesting differences between the classic peg in hole scenario vs. more complex navigational scenarios like the corner.





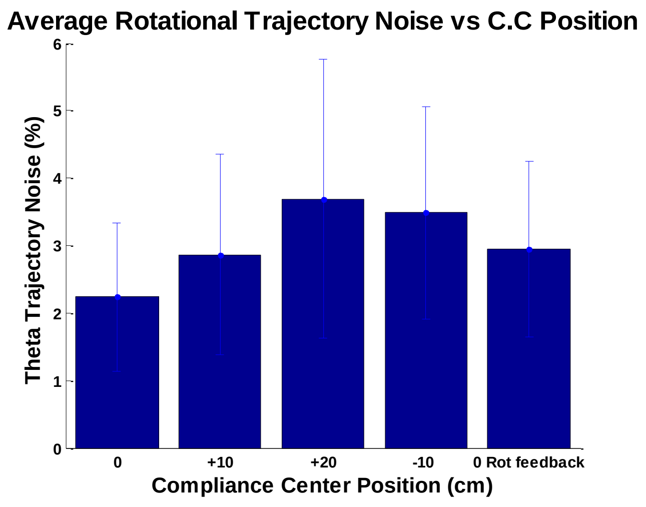


Figure 5.13: Average and standard deviation of noise in trajectories.

5.3 Survey

A survey was done with all the human subjects with questions asking their experience on driving an omnidirectional robot using a 3 DOF joystick with and without haptic feedback. This survey was on a scale of 1 to 5 with 1 being very poor and 5 being very good. The survey was mainly focused on user joystick driving experience in terms of collision avoidance and comfort. In Figure 5.15, it can be seen that user rated driving with 3-DOF force-feedback is better than with no feedback and zero rotational feedback. The average rating of driving with 3-DOF force-feedback is 4.75 while with no force-feedback is 1.87 and is 3.75 with zero rotational feedback.

The last question of survey asked users to pick the best algorithm out of the above three, i.e, with 3-DOF force-feedback, no force-feedback, and zero rotational feedback. 87.5 % said the trial with 3-DOF force-feedback was the best and 12.5 % said the trial with zero rotational feedback was best.

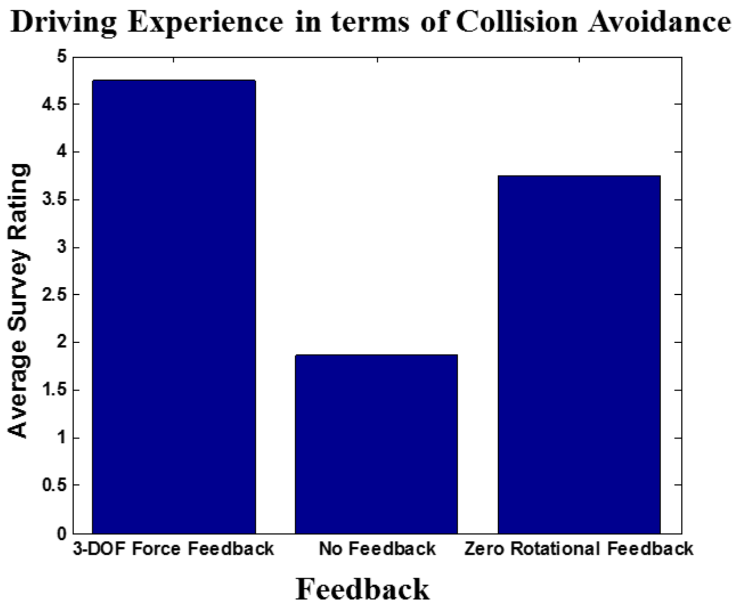


Figure 5.14: Survey of joystick driving experience in terms of collision avoidance.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This research is the first attempt to use true omnidirectional 3-DOF (degree of freedom) force-feedback to provide navigational assistance for a human to drive an omnidirectional vehicle. While 2-DOF force-feedback has been used in a limited capacity for obstacle avoidance on omnidirectional vehicles, this is the first research to include a third rotational axis of force-feedback, and to use it to guide a driver along planar collision, avoiding trajectories with a natural coordination of orientation. This research successfully accomplished use of a novel omnidirectional haptic device and force-feedback strategies to guide operators drive omnidirectional robots along collision-avoiding trajectories in an environment with obstacles. This is the first experiment to quantify the ability of omnidirectional force-feedback to improve omnidirectional driving performance and driver experience in real time scenario.

The primary aim of this research was to improve the intuitive control and embodiment of omnidirectional robots to optimize the driving performance of human operators. Omnidirectional robots could be used in operations requiring situational awareness as in search and rescue operations, first response, and law enforcement [50]. They also have lot of scope in industrial environments, for example, in bridges, tunnels, pipelines, and power plant boilers inspections [51]. Omni robots could be used in military operations too, such as for detection of unexploded ordinance or for carrying weight of soldiers in confined spaces. However, omnidirectional robots are hard to control and embodiment to the driver is not easy, as the driver has to control the x, y, and θ DOF from a remote location merely on the video feed from a camera (usually a single camera), due to the limited data bandwidth. Also, in most of the cases, complete autonomy is not preferred. Therefore, this research provides solutions to those control challenges posed by the limitations of intuitive control on omnidirectional robots, so the user could be assisted/encouraged to navigate on the obstacle free path.

The omnidirectional feedback law design worked as expected in terms of avoiding collisions on the path as well as improving the smoothness of the trajectory of the robot. From the results in previous sections, it can be concluded that by using omnidirectional haptic feedback collisions can be drastically improved. Significant improvements in smoothness of trajectories were also observed by the use of haptic feedback. Furthermore, it was shown that smoothness of trajectories was maximized by placing the compliance center at the center of the robot, an interesting result that is in contrast to more traditional strategies (associated with peg in hole scenarios) of placing the compliance center at the front of the robot.

6.2 Future Work

In the near future, continued research should focus on implementing these algorithms on a real omnidirectional robot instead of simulation. The existing Nexus robot could be used with improvements such as a LIDAR sensor to map the physical environment and a small onboard computer with ROS for faster processing. This onboard computer can communicate a LIDAR distance map over Wi-Fi to the computer connected to the joystick. The joystick setup can be used as is.

This research could also be used to impact the lives of the 1.3 % of our population [52]-[55] (and growing) that is wheelchair-bound. Omnidirectional powered-wheelchairs have the potential to increase the mobility and independence of the disabled, which are key factors in maintaining quality of life. New commercially available robots such as the Segway 440 Omni have the potential to jumpstart the availability of such wheelchairs. However, omnidirectional powered wheelchairs present a challenge to control, as operators must coordinate forward, lateral, and rotational motion. This research also seeks for intuitive and comfortable driving of omnidirectional wheelchairs, providing navigational guidance, while still allowing the driver to be in control. In addition, we can use the independent rotation of the omnidirectional wheelchair to assist occupants to maintain a visual connection with targeted people or television screens while they move about, restoring their sense of social connectivity.

The scope of this research could also later be extended to control of quadcopters using omnidirectional force-feedback as used in omnidirectional robots, with minor changes in the algorithms and approach. Control of quadcopters also poses some similar challenges as omnidirectional robots, for example: lack of intuitive control, lack of embodiment, and limited data bandwidth. Thus, similar schemes could be used to improve intuitive control on quadcopters.

APPENDIX A

SURVEY

Please answer questions below on the scale of 1 to 5,

(1) Very Poor, (2) Poor, (3) Average, (4) Good and (5) Very Good

1. How would you rate your Joystick driving experience **with** Force-feedback in terms of Collision Avoidance?

   1 2 3 4 5

2. How would you rate your Joystick driving experience **with** Force-feedback in terms of Comfort level?

   1 2 3 4 5

3. How would you rate your Joystick driving experience **without** Force-feedback in terms of Collision Avoidance?

   1 2 3 4 5

4. How would you rate your Joystick driving experience **without** Force-feedback in terms of Comfort level?

   1 2 3 4 5

5. How would you rate your Joystick driving experience when the **Compliance Center** was moved to **front of Robot,** in terms of Collision Avoidance?

   1 2 3 4 5

6. How would you rate your Joystick driving experience when the **Compliance Center** was moved to **front of Robot,** in terms of Comfort Level?

   1 2 3 4 5

7. How would you rate your Joystick driving experience with **Zero Rotational Feedback** in terms of Collision Avoidance?

   1 2 3 4 5

8. How would you rate your Joystick driving experience **Zero Rotational Feedback** in terms of Comfort level?

   1 2 3 4 5

9. Out of all 4 different set of trials below, which one do you think was best:

   ☐ With Force-feedback ☐ Without Force-feedback

   ☐ Zero Rotational Feedback

10. How good are your Video game playing skills?

    1 2 3 4 5

Any other Comments:

APPENDIX B

PROGRAMMING CODE

Arduino Main Code

```
/*

Joystick Force-feedback

Author: Rajat Tyagi
Date: 10/15/2016

This Program reads joystick encoder values using Encoder.h library and send them to
VREP Simulation via ROS Node as user Input
It also receives Force-feedback data from VREP via another ROS Node and uses i to
produce haptic Feeedback on Joystick
This program also implements the force-feedback


Interrupt rate 100 KHz
Interrupt pins: Encoder-X (20,21); Encoder-Y(18,19); Encoder-Theta(2,3);



*/
#include <ros.h>
#include <std_msgs/String.h>

#include <Encoder.h>
#include <JoystickEncoderRos.h>
#include <JoystickFeedbackRos.h>

ros::NodeHandle nh;


std_msgs::String str_msg;
String msg;
```

```
float force[]={0,0,0};

void messageCb( std_msgs::String& toggle_msg)
  {
   msg = String(toggle_msg.data);
  // Serial3.print(msg);


int commaIndex = msg.indexOf(',');

int secondCommaIndex = msg.indexOf(',', commaIndex+1);



String firstValue = msg.substring(0, commaIndex);
String secondValue = msg.substring(commaIndex+1, secondCommaIndex);
String thirdValue = msg.substring(secondCommaIndex+1);


force[0] = firstValue.toFloat();
force[1]=  secondValue.toFloat();
force[2] = thirdValue.toFloat();

}

  ros::Subscriber<std_msgs::String> forceSub("feedForce", &messageCb );

  ros::Publisher chatter("omniVel", &str_msg);

String outVel;
char outVelChar[25];
int inByte;
const int len=12;
int sonarVals[len];
int i=0;                   //dataBuffer index
bool record=0;             //Flag to start storing data
bool setControl;           //Flag to set control values

unsigned long lastT;

int joyPos[]={0,0,0};
float joyVel[]={0,0,0};



void setup()
```

```
{

nh.initNode();
nh.advertise(chatter);
nh.subscribe(forceSub);

//Serial.begin(9600);
//Serial3.begin(9600);

initEncoder(); //Initialize encoder parameters and change frequency of PWM output
initFeedback(); //Initialize all pins for force-feedback
}

void loop() {

unsigned long nowT = millis();
double timeDiff = (double) (nowT - lastT);

//Serial3.print(force[2]);
encoderPos(joyPos, joyVel);

forceFeedback(joyPos,force, joyVel, timeDiff);

lastT = nowT;

   outVel +=  joyVel[0];
   outVel += ",";
   outVel += joyVel[1];
   outVel += ",";
   outVel += joyVel[2];
   //outVel += "\n";

  outVel.toCharArray(outVelChar,25);


  str_msg.data = outVelChar;
  chatter.publish( &str_msg );
  nh.spinOnce();

   outVel  = String("");
lastT = nowT;

}
```

Joystick Encoder.h

```
/*
JoystickEncoderRos.h

Author: Rajat Tyagi
Date:   4/28/2016

This code is a support library for joystickforcefeedback.ino, and should be saved in
it's own directory in your Arduino Libraries folder.

This library provides functions to interact Encoders.
*/

#ifndef JoystickEncoderRos_h
#define JoystickEncoderRos_h


#include "Arduino.h"
//Initialize Encoders

void initEncoder();



void encoderPos(int *encPos, float *joyVel);


#endif
```

Joystick Encoder.cpp

```
/*

JoystickEncoder.cpp

Author: Rajat Tyagi
Date: 1/15/2016

This Program reads joystick encoder values using Encoder.h library

Interrupt rate 100 KHz
nterrupt pins: Encoder-X (20,21); Encoder-Y(18,19); Encoder-Theta(2,3);

*/
```

```
#include "JoystickEncoderRos.h"
#include <Encoder.h>

Encoder encoder_x(20, 21);
Encoder encoder_y(18, 19);
Encoder encoder_Q(3,2);
String outVelocity;

void initEncoder()
 {

int Eraser = 7;     //This is 111 in binary and is used as an eraser
TCCR2B &=~Eraser;   // This operation (AND plus NOT), set the three bits in TCCR3B
and TCCR4B to 0
TCCR1B &=~Eraser;
TCCR4B &=~Eraser;

int myPrescaler = 1;  // This could be number [1,6], 1 corresponds 001 in binary and sets
prescaler for frequency 31000 HZ
TCCR2B |= myPrescaler; // This changes last 3 bits in TCCR3B with 001
TCCR1B |= myPrescaler;
TCCR4B |= myPrescaler; // This changes last 3 bits in TCCR4B with 001

// For more information visit http://forum.arduino.cc/index.php?topic=72092.0
 encoder_x.write(0);
 encoder_y.write(0);
 encoder_Q.write(0);

 outVelocity = String("");

}

void encoderPos(int *encPos, float *joyVel)
{

int int_mask=256;
int trunc_x=0;
int trunc_y=0;
int trunc_Q=0;

int dec_x=0;
int dec_y=0;
int dec_Q=0;

double position_x  = 0;
double position_y = 0;
double position_Q = 0;
```

```
  double new_x_enc, new_y_enc, new_Q_enc;
  double new_x, new_y, new_Q;
   new_x = encoder_x.read();
   new_y = encoder_y.read();
   new_Q = encoder_Q.read();

   new_x_enc=new_x;
   new_y_enc=new_y;
   new_Q_enc=new_Q;



if (new_x>300)
{
  new_x=300;
}


if (new_x<-300)
{
  new_x=-300;
}

if (new_y>300)
{
  new_y=300;
}

if (new_y<-300)
{
  new_y=-300;
}

if (new_Q>1440)
{
  new_Q=1440;
}

if (new_Q<-1440)
{
  new_Q=-1440;
}

    position_x = new_x*0.003;  //Converts encoder count to linear velocity of omni robot
in X-direction
    position_y = new_y*0.003;  //Converts encoder count to linear velocity of omni robot
```

```
in Y-direction
    position_Q = new_Q*0.012;  //Converts encoder count to Rotational velocity of omni
robot



// put a deadzone in X, Y and theta for better control
  if (position_x > -0.2 && position_x < 0.2)   // Previous was +-0.1
  {
    position_x = 0;
  }

  if (position_y> -0.2 && position_y < 0.2)  // Previous was +-0.1
  {
    position_y = 0;
  }

  if (position_Q < -3)
  {
    position_Q = -3;
  }

  if ( position_Q > 3)
  {
    position_Q = 3;
  }

  if ( position_Q < 0.5 && position_Q > - 0.5  )
  {
    position_Q = 0;
  }



joyVel[0] = position_x;
joyVel[1] = position_y;
joyVel[2] = position_Q;

   // Serial.print(position_x);
   // Serial.print("   ");
   // Serial.print(position_y);
   // Serial.print("   ");
   // Serial.println(position_Q);

//Convert velocity to integer and decimal part to send via xbee, this will be converted
back when recived at robot
```

```
trunc_x=(int)position_x;
trunc_y=(int)position_y;
trunc_Q=(int)position_Q;

dec_x=(position_x-trunc_x)*100;
dec_y=(position_y-trunc_y)*100;
dec_Q=(position_Q-trunc_Q)*100;


// Peprare array to send via xbee, containing velocities with start and end byte
// An int mask is added to convert all values from 0 to 255

int array_transmit[] = {-127+int_mask, trunc_x+int_mask, dec_x+int_mask,
trunc_y+int_mask, dec_y+int_mask, trunc_Q+int_mask, dec_Q+int_mask, -
128+int_mask};


// Send Data via Serial 3 to xbee

  // Serial.print(joyVel[0]);
  // Serial.print(" ");
  // Serial.print(joyVel[1]);
  // Serial.print(" ");
  // Serial.println(joyVel[2]);

  // outVelocity +=  joyVel[0];
  // outVelocity += ",     ";
  // outVelocity += joyVel[1];
  // outVelocity += ",     ";
  // outVelocity += joyVel[2];
  // outVelocity += "\n";
  //   str_msg.data = outVelocity;
  //Serial.print(outVelocity);
  outVelocity  = String("");

//Serial.println();

   encPos[0] = new_x_enc;
   encPos[1] = new_y_enc;
   encPos[2] = new_Q_enc;

}
```

JoystickFeedbackRos.h

```
/*
JoystickFeedbackRos.h

Author: Rajat Tyagi
Date:   4/28/2016

This code is a support library for JoystickForceFeedback_ROS.ino, and should be saved
in
it's own directory in your Arduino Libraries folder.

This library provides Force-feedback commands for joystick
*/

#ifndef JoystickFeedbackRos_h
#define JoystickFeedbackRos_h

#include "Arduino.h"
//Initialize Encoders

void initFeedback();

int saturate(int value, int satVal);

void zeroFeedbackLaw(int *joyPos, double timed);

void writePwm(float matVal, int axisPos);

void feedbackLaw(float *forceVal, float *joyVel);

void forceFeedback(int *joyPos, float *forceVal, float *joyVel, double timed);


#endif
```

JoystickFeedbackRos.cpp

```
/*

JoystickFeedbackRos.cpp

Author: Rajat Tyagi
```

```
Date: 4/28/2016

This library provides Force-feedback commands for joystick


*/




#include "JoystickFeedbackRos.h"
#include <math.h>



float pwmVal[]={0,0,0,0,0,0};
float pwm_max=240;
float slope= 12;
float int_error=0;
float error_def_prev=0;
float GainP= 0.2;
float GainD= 2;
float GainI=0;
float pwm_def=0;
String outBuffer;
int cycleCount = 0;

// below parameters are for joystick zero position PD contoller
float posActual[3];
float lastPos[3];
float posDesired[]={0,0,0};
float controlEffort[3];
float lastError[3];


float Kp = 0.7;    // P Gain for Zero Position Feedback, old Gain = 0.7;
float Kd = 42;   // D Gain for Zero Position Feedback, old Gain = 42;


float error;
float diff;
float zeroPos[6];
float joyAngle[]={0,0,0};
const float pi = 3.14;


//int timed = 8;   // approx loop time in ms from main program, used for calculating
```

damping feedback

```
void initFeedback()
 {

// Sets pimode to output
outBuffer = String("");
pinMode(8,OUTPUT);
pinMode(9,OUTPUT);
pinMode(6,OUTPUT);
pinMode(7,OUTPUT);
pinMode(10,OUTPUT);
pinMode(11,OUTPUT);

}


//Saturate to a threshold

int saturate(int value, int satVal)
{
  if(value > satVal)
    return satVal;
  else if (value < -satVal)
    return -satVal;
  else
    return value;
}


void zeroFeedbackLaw(int *joyPos,double timed) // This function brings joystick to zero
position using PD contoller
{


joyAngle[0] = (float)joyPos[0] / 12;                    // Calculates angle of Joystick from
encoder position
joyAngle[1] = (float)joyPos[1] / 15;
joyAngle[2] = (float)joyPos[2] * 360/1440;

for (int i=0; i<3; i++)
  {
    //Calculate wheel speeds [rpm]: 1 tick/10ms=1.95 rpm
 joyPos[i] = (float)joyPos[i]*360/1440 ;
  }
 for (int i=0; i<3; i++)
 {
  error = posDesired[i] - joyPos[i]; //Proportional term
  diff= (float)((error-lastError[i])/timed);          //Derivative term


  controlEffort[i]=Kp*error + Kd*diff;

if(i==2)
  {
    controlEffort[i]=Kp*error*1.9 + Kd*diff*1.1;
  }
  //controlEffort[i]=saturate(controlEffort[i], 255); //Saturate PMW to +/- 255
  lastError[i]= error;
 }



 for (int i=0; i<6 ;++i)
 {
 zeroPos[i]= 0;
 }


float alphaGrav_X = 100;
float alphaGrav_Y = 70;


// float alphaGrav_X = 100;

  //Saves Control Effort for Zero Position in all directions in an array (See header of this
code for detail)

        if (controlEffort[0] >= 0)              // From - X to +X
        {
          zeroPos[0] = controlEffort[0] - alphaGrav_X * sin((joyAngle[0])  *pi/180);
        }

        else if (controlEffort[0] < 0)          // From + X to +X
        {
          zeroPos[1] = - controlEffort[0] + alphaGrav_X * sin(joyAngle[0] *pi/180);
        }

        if (controlEffort[1] >= 0)            // From - Y to +Y
        {
          zeroPos[2] = + controlEffort[1]  - alphaGrav_Y * sin((joyAngle[1])  *pi/180);
        }

        if (controlEffort[1] < 0)            // From + Y to -Y
        {
          zeroPos[3] = - controlEffort[1] + alphaGrav_Y * sin(joyAngle[1] *pi/180);;
        }

        if (controlEffort[2] >= 0)           // -Theta to + Theta
        {
          zeroPos[4] =  controlEffort[2];
        }

        if (controlEffort[2] < 0)            // +Theta to - Theta
        {
          zeroPos[5] = - controlEffort[2];
        }

}

// This function calculates and writes pwm value to amplifier

void writePwm(float matVal, int axisPos)
{

double write1,write2 = 0;

 if (matVal >0)
 {
  write1 = matVal;
  write2 = 0;
 }

 else
 {
  write2 = -matVal;
  write1 = 0;
 }

float sendPositive = constrain(write1 + zeroPos [axisPos*2],0,255);
float sendNegative = constrain(write2 + zeroPos [axisPos*2 + 1],0,255);

// int sendPositive = constrain(write1 + zeroPos [axisPos*2],0,255);
// int sendNegative = constrain(write2 + zeroPos [axisPos*2 + 1],0,255);

analogWrite(axisPos*2 + 6, sendPositive);
analogWrite((axisPos*2 + 6)+1, sendNegative);
```

```
//analogWrite(axisPos*2 + 6, 255);
//analogWrite((axisPos*2 + 6)+1, 0);



}

// This function calculates feedback based on sensor values
// The Feedback Law have to be implemented in this function

void feedbackLaw(float *forceVal, float *joyVel)
{



writePwm(forceVal[0], 0);
writePwm(forceVal[1], 1);
writePwm(forceVal[2], 2);

// writePwm(0, 0);
// writePwm(0, 1);
// writePwm(0, 2);



}


void forceFeedback(int *joyPos, float *forceVal, float *joyVel, double timed)

{

// Serial.print(joyPos[0]);
// Serial.print("  ");
// Serial.print(joyPos[1]);
// Serial.print("  ");
// Serial.println(joyPos[2]);


zeroFeedbackLaw(joyPos, timed);
feedbackLaw(forceVal, joyVel);

}
```

VREP Youbot Code

```
--- This example script is non-threaded (executed at each simulation pass)
-- The functionality of this script (or parts of it) could be implemented
-- in an extension module (plugin) and be hidden. The extension module could
-- also allow connecting to and controlling the real robot.

-- Rajat Tyagi
--10/16/2016

-- THis Script is to run Youbot in VREP simulation mode in ROS,
-- It needs the Huloyu script along with it



function string:split( inSplitPattern, outResults )
  if not outResults then
    outResults = { }
  end
  local theStart = 1
  local theSplitStart, theSplitEnd = string.find( self, inSplitPattern, theStart )
  while theSplitStart do
    table.insert( outResults, string.sub( self, theStart, theSplitStart-1 ) )
    theStart = theSplitEnd + 1
    theSplitStart, theSplitEnd = string.find( self, inSplitPattern, theStart )
  end
  table.insert( outResults, string.sub( self, theStart ) )
  return outResults
end


function joyStickMessage_callback(msg)

--simAddStatusbarMessage = string.split( msg.data, "," )
    local myString = msg.data

    local myVel = myString:split(",")
     VelsX = 0.4 * tonumber(myVel[1])
     VelsY = 0.4 * tonumber(myVel[2])
     VelsQ = 0.4 * tonumber(myVel[3])


if(VelsX > 0 or VelsY > 0 or VelsQ > 0) and timeFlag == 0 then
    timerStart = simGetSimulationTime()
    timeFlag = 1
   -- print("asadsadfghdfs")
end
```

```
 end

if (sim_call_type==sim_childscriptcall_initialization) then
   -- First time we execute this script.
   count = 0
   collsRed =0
   collsBlue =0
   collsGreen =0
   collisionNumber = 1
   collisionR  = 0
   collisionB  = 0
   collisionG  = 0
   timeFlag = 0

  objPosXold = 0.62

    lastTime = 0
   currentTime = 0
     --Prepare initial values and retrieve handles:
   wheelJoints={-1,-1,-1,-1} -- front left, rear left, rear right, front right
   wheelJoints[1]=simGetObjectHandle('rollingJoint_fl')
   wheelJoints[2]=simGetObjectHandle('rollingJoint_rl')
   wheelJoints[3]=simGetObjectHandle('rollingJoint_rr')
   wheelJoints[4]=simGetObjectHandle('rollingJoint_fr')


   youBot=simGetObjectHandle('youBot')
   rect=simGetObjectHandle('ME_Platfo2_sub1')

 --   wheel_rl=simGetObjectHandle('swedishWheel_rl')
--   wheel_rr=simGetObjectHandle('swedishWheel_rr')
 --   wheel_fl=simGetObjectHandle('swedishWheel_fl')
 --   wheel_fr=simGetObjectHandle('swedishWheel_fr')


   wheel_rl=simGetObjectHandle('wheel_respondable_rl')
   wheel_rr=simGetObjectHandle('wheel_respondable_rr')
   wheel_fl=simGetObjectHandle('wheel_respondable_fl')
   wheel_fr=simGetObjectHandle('wheel_respondable_fr')



  -- inter_rl=simGetObjectHandle('wheel_respondable_rl')
   -- wheel_rr=simGetObjectHandle('swedishWheel_rr')
   -- inter_fl=simGetObjectHandle('wheel_respondable_fl')
  --  wheel_fr=simGetObjectHandle('swedishWheel_fr')
```

```
youBotRef=simGetObjectHandle('youBot_ref')

wall=simGetObjectHandle('80cmHighWall100cm_visible')
wall0=simGetObjectHandle('80cmHighWall100cm_visible0')
wall1=simGetObjectHandle('80cmHighWall100cm_visible1')
wall2=simGetObjectHandle('80cmHighWall100cm_visible2')
wall3=simGetObjectHandle('80cmHighWall50cm_visible11')
wall4=simGetObjectHandle('80cmHighWall100cm_visible4')
wall5=simGetObjectHandle('80cmHighWall100cm_visible5')
wall6=simGetObjectHandle('80cmHighWall100cm_visible6')
wall7=simGetObjectHandle('80cmHighWall100cm_visible7')
wall8=simGetObjectHandle('80cmHighWall100cm_visible8')

wall9=simGetObjectHandle('80cmHighWall50cm_visible')
wall10=simGetObjectHandle('80cmHighWall50cm_visible0')
wall11=simGetObjectHandle('80cmHighWall50cm_visible1')
wall12=simGetObjectHandle('80cmHighWall50cm_visible2')
wall13=simGetObjectHandle('80cmHighWall50cm_visible3')
wall14=simGetObjectHandle('80cmHighWall50cm_visible4')
wall15=simGetObjectHandle('80cmHighWall50cm_visible5')
wall16=simGetObjectHandle('80cmHighWall50cm_visible6')
wall17=simGetObjectHandle('80cmHighWall50cm_visible7')
wall18=simGetObjectHandle('80cmHighWall50cm_visible8')
wall19=simGetObjectHandle('80cmHighWall50cm_visible9')
wall20=simGetObjectHandle('80cmHighWall50cm_visible10')

wall21=simGetObjectHandle('80cmHighWall200cm_visible')
wall22=simGetObjectHandle('80cmHighWall200cm_visible0')
wall23=simGetObjectHandle('80cmHighWall200cm_visible1')
wall24=simGetObjectHandle('80cmHighWall200cm_visible2')
wall25=simGetObjectHandle('80cmHighWall200cm_visible3')
wall26=simGetObjectHandle('80cmHighWall200cm_visible4')
wall27=simGetObjectHandle('80cmHighWall200cm_visible5')
wall28=simGetObjectHandle('80cmHighWall200cm_visible6')
wall29=simGetObjectHandle('80cmHighWall200cm_visible7')
wall30=simGetObjectHandle('80cmHighWall200cm_visible8')
wall31=simGetObjectHandle('80cmHighWall200cm_visible9')
wall32=simGetObjectHandle('80cmHighWall200cm_visible10')
wall33=simGetObjectHandle('80cmHighWall200cm_visible11')
wall34=simGetObjectHandle('80cmHighWall200cm_visible12')
wall35=simGetObjectHandle('80cmHighWall200cm_visible13')
wall36=simGetObjectHandle('80cmHighWall200cm_visible14')
wall37=simGetObjectHandle('80cmHighWall200cm_visible15')
wall38=simGetObjectHandle('80cmHighWall200cm_visible16')
wall39=simGetObjectHandle('80cmHighWall200cm_visible17')
wall40=simGetObjectHandle('80cmHighWall200cm_visible18')
```

```
   wall41=simGetObjectHandle('80cmHighWall200cm_visible19')
   wall42=simGetObjectHandle('80cmHighWall200cm_visible20')


   --block=simGetObjectHandle('ConcretBlock')
   block1=simGetObjectHandle('ConcretBlock0')
   blockG=simGetObjectHandle('ConcretBlock#0')
   blockB=simGetObjectHandle('ConcretBlock#1')
   blockR=simGetObjectHandle('ConcretBlock#2')


   target=simGetObjectHandle('youBot_positionTarget')


   VelsX = 0
   VelsY = 0
   VelsQ = 0




 Jinv11 = 20 ; Jinv12 = -20; Jinv13 = -5.8;
 Jinv21 = 20 ; Jinv22 = 20 ; Jinv23 = -5.8;
 Jinv31 = -20; Jinv32 = 20 ; Jinv33 = -5.8;
 Jinv41 = -20; Jinv42 = -20; Jinv43 = -5.8;


  --forwBackVelRange={-240*math.pi/180,240*math.pi/180}  -- min and max wheel
rotation vel. for backward/forward movement
  -- leftRightVelRange={-240*math.pi/180,240*math.pi/180} -- min and max wheel
rotation vel. for left/right movement
  -- rotVelRange={-240*math.pi/180,240*math.pi/180}       -- min and max wheel
rotation vel. for left/right rotation movement

   sub=simExtRosInterface_subscribe('/omniVel', 'std_msgs/String',
'joyStickMessage_callback')




 --simAddStatusbarMessage('asdsf')
end

if (sim_call_type==sim_childscriptcall_cleanup) then
```

```
end

if (sim_call_type==sim_childscriptcall_actuation) then

-----------------------------------------------------------------------------------

  compCenter=simGetScriptSimulationParameter(sim_handle_self,'complianceCenter')

-----------------------------------------------------------------------------------

   currentTime=simGetSimulationTime()


  -- compC = simGetFloatSignal("compC")
   simSetFloatSignal("VeloX",VelsX)
   simSetFloatSignal("VeloY",VelsY)
   simSetFloatSignal("VeloQ",VelsQ)
   simSetFloatSignal("Time",currentTime)
   simSetFloatSignal("compC",compCenter)

--   simSetFloatSignal("yCord",objPosY)

   VelsY = VelsY + VelsQ * compCenter
   omegaDesiredFL=(Jinv11*VelsX+Jinv12*VelsY+Jinv13*VelsQ);
   omegaDesiredRL=(Jinv21*VelsX+Jinv22*VelsY+Jinv23*VelsQ);
   omegaDesiredRR=(Jinv31*VelsX+Jinv32*VelsY+Jinv33*VelsQ);
   omegaDesiredFR=(Jinv41*VelsX+Jinv42*VelsY+Jinv43*VelsQ);
   --simAddStatusbarMessage(omegaDesiredFR)


   simSetJointTargetVelocity(wheelJoints[1],omegaDesiredFL)
   simSetJointTargetVelocity(wheelJoints[2],omegaDesiredRL)
   simSetJointTargetVelocity(wheelJoints[3],-omegaDesiredRR)
   simSetJointTargetVelocity(wheelJoints[4],-omegaDesiredFR)
   --simAddStatusbarMessage('asdsf')

  if ((simCheckCollision(wheel_rr,wall) ==1) or (simCheckCollision(wheel_rl,wall) ==1)
or (simCheckCollision(wheel_fr,wall) ==1) or (simCheckCollision(wheel_fl,wall) ==1)
or (simCheckCollision(youBot,wall) ==1)) then
   colls = 1
  elseif ((simCheckCollision(wheel_rr,wall0) ==1) or
(simCheckCollision(wheel_rl,wall0) ==1) or (simCheckCollision(wheel_fr,wall0) ==1)
or (simCheckCollision(wheel_fl,wall0) ==1) or (simCheckCollision(youBot,wall0) ==1))
then
      colls = 1
  elseif ((simCheckCollision(wheel_rr,wall1) ==1) or
(simCheckCollision(wheel_rl,wall1) ==1) or (simCheckCollision(wheel_fr,wall1) ==1)
or (simCheckCollision(wheel_fl,wall1) ==1) or (simCheckCollision(youBot,wall1) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall2) ==1) or
(simCheckCollision(wheel_rl,wall2) ==1) or (simCheckCollision(wheel_fr,wall2) ==1)
or (simCheckCollision(wheel_fl,wall2) ==1) or (simCheckCollision(youBot,wall2) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall3) ==1) or
(simCheckCollision(wheel_rl,wall3) ==1) or (simCheckCollision(wheel_fr,wall3) ==1)
or (simCheckCollision(wheel_fl,wall3) ==1) or (simCheckCollision(youBot,wall3) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall4) ==1) or
(simCheckCollision(wheel_rl,wall4) ==1) or (simCheckCollision(wheel_fr,wall4) ==1)
or (simCheckCollision(wheel_fl,wall4) ==1) or (simCheckCollision(youBot,wall4) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall5) ==1) or
(simCheckCollision(wheel_rl,wall5) ==1) or (simCheckCollision(wheel_fr,wall5) ==1)
or (simCheckCollision(wheel_fl,wall5) ==1) or (simCheckCollision(youBot,wall5) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall6) ==1) or
(simCheckCollision(wheel_rl,wall6) ==1) or (simCheckCollision(wheel_fr,wall6) ==1)
or (simCheckCollision(wheel_fl,wall6) ==1) or (simCheckCollision(youBot,wall6) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall7) ==1) or
(simCheckCollision(wheel_rl,wall7) ==1) or (simCheckCollision(wheel_fr,wall7) ==1)
or (simCheckCollision(wheel_fl,wall7) ==1) or (simCheckCollision(youBot,wall7) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall8) ==1) or
(simCheckCollision(wheel_rl,wall8) ==1) or (simCheckCollision(wheel_fr,wall8) ==1)
or (simCheckCollision(wheel_fl,wall8) ==1) or (simCheckCollision(youBot,wall8) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall9) ==1) or
(simCheckCollision(wheel_rl,wall9) ==1) or (simCheckCollision(wheel_fr,wall9) ==1)
or (simCheckCollision(wheel_fl,wall9) ==1) or (simCheckCollision(youBot,wall9) ==1))
then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall10) ==1) or
(simCheckCollision(wheel_rl,wall10) ==1) or (simCheckCollision(wheel_fr,wall10)
==1) or (simCheckCollision(wheel_fl,wall10) ==1) or
(simCheckCollision(youBot,wall10) ==1)) then
```

```
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall11) ==1) or
(simCheckCollision(wheel_rl,wall11) ==1) or (simCheckCollision(wheel_fr,wall11) ==1)
or (simCheckCollision(wheel_fl,wall11) ==1) or (simCheckCollision(youBot,wall11)
==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall12) ==1) or
(simCheckCollision(wheel_rl,wall12) ==1) or (simCheckCollision(wheel_fr,wall12)
==1) or (simCheckCollision(wheel_fl,wall12) ==1) or
(simCheckCollision(youBot,wall12) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall13) ==1) or
(simCheckCollision(wheel_rl,wall13) ==1) or (simCheckCollision(wheel_fr,wall13)
==1) or (simCheckCollision(wheel_fl,wall13) ==1) or
(simCheckCollision(youBot,wall13) ==1)) then
   colls = 1


  elseif ((simCheckCollision(wheel_rr,wall14) ==1) or
(simCheckCollision(wheel_rl,wall14) ==1) or (simCheckCollision(wheel_fr,wall14)
==1) or (simCheckCollision(wheel_fl,wall14) ==1) or
(simCheckCollision(youBot,wall14) ==1)) then
        colls = 1
  elseif ((simCheckCollision(wheel_rr,wall15) ==1) or
(simCheckCollision(wheel_rl,wall15) ==1) or (simCheckCollision(wheel_fr,wall15)
==1) or (simCheckCollision(wheel_fl,wall15) ==1) or
(simCheckCollision(youBot,wall15) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall16) ==1) or
(simCheckCollision(wheel_rl,wall16) ==1) or (simCheckCollision(wheel_fr,wall16)
==1) or (simCheckCollision(wheel_fl,wall16) ==1) or
(simCheckCollision(youBot,wall16) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall17) ==1) or
(simCheckCollision(wheel_rl,wall17) ==1) or (simCheckCollision(wheel_fr,wall17)
==1) or (simCheckCollision(wheel_fl,wall17) ==1) or
(simCheckCollision(youBot,wall17) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall18) ==1) or
(simCheckCollision(wheel_rl,wall18) ==1) or (simCheckCollision(wheel_fr,wall18)
==1) or (simCheckCollision(wheel_fl,wall18) ==1) or
(simCheckCollision(youBot,wall18) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall19) ==1) or
(simCheckCollision(wheel_rl,wall19) ==1) or (simCheckCollision(wheel_fr,wall19)
==1) or (simCheckCollision(wheel_fl,wall19) ==1) or
(simCheckCollision(youBot,wall19) ==1)) then
```

```
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall20) ==1) or
(simCheckCollision(wheel_rl,wall20) ==1) or (simCheckCollision(wheel_fr,wall20)
==1) or (simCheckCollision(wheel_fl,wall20) ==1) or
(simCheckCollision(youBot,wall20) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall21) ==1) or
(simCheckCollision(wheel_rl,wall21) ==1) or (simCheckCollision(wheel_fr,wall21)
==1) or (simCheckCollision(wheel_fl,wall21) ==1) or
(simCheckCollision(youBot,wall21) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall22) ==1) or
(simCheckCollision(wheel_rl,wall22) ==1) or (simCheckCollision(wheel_fr,wall22)
==1) or (simCheckCollision(wheel_fl,wall22) ==1) or
(simCheckCollision(youBot,wall22) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall23) ==1) or
(simCheckCollision(wheel_rl,wall23) ==1) or (simCheckCollision(wheel_fr,wall23)
==1) or (simCheckCollision(wheel_fl,wall23) ==1) or
(simCheckCollision(youBot,wall23) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall24) ==1) or
(simCheckCollision(wheel_rl,wall24) ==1) or (simCheckCollision(wheel_fr,wall24)
==1) or (simCheckCollision(wheel_fl,wall24) ==1) or
(simCheckCollision(youBot,wall24) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall25) ==1) or
(simCheckCollision(wheel_rl,wall25) ==1) or (simCheckCollision(wheel_fr,wall25)
==1) or (simCheckCollision(wheel_fl,wall25) ==1) or
(simCheckCollision(youBot,wall25) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall26) ==1) or
(simCheckCollision(wheel_rl,wall26) ==1) or (simCheckCollision(wheel_fr,wall26)
==1) or (simCheckCollision(wheel_fl,wall26) ==1) or
(simCheckCollision(youBot,wall26) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall27) ==1) or
(simCheckCollision(wheel_rl,wall27) ==1) or (simCheckCollision(wheel_fr,wall27)
==1) or (simCheckCollision(wheel_fl,wall27) ==1) or
(simCheckCollision(youBot,wall27) ==1)) then
   colls = 1


  elseif ((simCheckCollision(wheel_rr,wall28) ==1) or
(simCheckCollision(wheel_rl,wall28) ==1) or (simCheckCollision(wheel_fr,wall28)
==1) or (simCheckCollision(wheel_fl,wall28) ==1) or
(simCheckCollision(youBot,wall28) ==1)) then
```

```
      colls = 1
  elseif ((simCheckCollision(wheel_rr,wall29) ==1) or
(simCheckCollision(wheel_rl,wall29) ==1) or (simCheckCollision(wheel_fr,wall29)
==1) or (simCheckCollision(wheel_fl,wall29) ==1) or
(simCheckCollision(youBot,wall29) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall30) ==1) or
(simCheckCollision(wheel_rl,wall30) ==1) or (simCheckCollision(wheel_fr,wall30)
==1) or (simCheckCollision(wheel_fl,wall30) ==1) or
(simCheckCollision(youBot,wall30) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall31) ==1) or
(simCheckCollision(wheel_rl,wall31) ==1) or (simCheckCollision(wheel_fr,wall31)
==1) or (simCheckCollision(wheel_fl,wall31) ==1) or
(simCheckCollision(youBot,wall31) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall32) ==1) or
(simCheckCollision(wheel_rl,wall32) ==1) or (simCheckCollision(wheel_fr,wall32)
==1) or (simCheckCollision(wheel_fl,wall32) ==1) or
(simCheckCollision(youBot,wall32) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall33) ==1) or
(simCheckCollision(wheel_rl,wall33) ==1) or (simCheckCollision(wheel_fr,wall33)
==1) or (simCheckCollision(wheel_fl,wall33) ==1) or
(simCheckCollision(youBot,wall33) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall34) ==1) or
(simCheckCollision(wheel_rl,wall34) ==1) or (simCheckCollision(wheel_fr,wall34)
==1) or (simCheckCollision(wheel_fl,wall34) ==1) or
(simCheckCollision(youBot,wall34) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall35) ==1) or
(simCheckCollision(wheel_rl,wall35) ==1) or (simCheckCollision(wheel_fr,wall35)
==1) or (simCheckCollision(wheel_fl,wall35) ==1) or
(simCheckCollision(youBot,wall35) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall36) ==1) or
(simCheckCollision(wheel_rl,wall36) ==1) or (simCheckCollision(wheel_fr,wall36)
==1) or (simCheckCollision(wheel_fl,wall36) ==1) or
(simCheckCollision(youBot,wall36) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall37) ==1) or
(simCheckCollision(wheel_rl,wall37) ==1) or (simCheckCollision(wheel_fr,wall37)
==1) or (simCheckCollision(wheel_fl,wall37) ==1) or
(simCheckCollision(youBot,wall37) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall38) ==1) or
(simCheckCollision(wheel_rl,wall38) ==1) or (simCheckCollision(wheel_fr,wall38)
==1) or (simCheckCollision(wheel_fl,wall38) ==1) or
(simCheckCollision(youBot,wall38) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall39) ==1) or
(simCheckCollision(wheel_rl,wall39) ==1) or (simCheckCollision(wheel_fr,wall39)
==1) or (simCheckCollision(wheel_fl,wall39) ==1) or
(simCheckCollision(youBot,wall39) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall40) ==1) or
(simCheckCollision(wheel_rl,wall40) ==1) or (simCheckCollision(wheel_fr,wall40)
==1) or (simCheckCollision(wheel_fl,wall40) ==1) or
(simCheckCollision(youBot,wall40) ==1)) then
    colls = 1
  elseif ((simCheckCollision(wheel_rr,wall41) ==1) or
(simCheckCollision(wheel_rl,wall41) ==1) or (simCheckCollision(wheel_fr,wall41)
==1) or (simCheckCollision(wheel_fl,wall41) ==1) or
(simCheckCollision(youBot,wall41) ==1)) then
   colls = 1
  elseif ((simCheckCollision(wheel_rr,wall42) ==1) or
(simCheckCollision(wheel_rl,wall42) ==1) or (simCheckCollision(wheel_fr,wall42)
==1) or (simCheckCollision(wheel_fl,wall42) ==1) or
(simCheckCollision(youBot,wall42) ==1)) then
   colls = 1

  else
    colls =0
  end



    if (colls == 1 and count==0) then
        a = "!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  Collision num !!!!!!!!!!!!!!!!!!!!!!"
        b = string.gsub(a, "num",collisionNumber)
        dialogHandle = simDisplayDialog('Drive away from
wall',b ,sim_dlgstyle_ok,false,nil,{0,0.5,0,0,0,0},{0.5,0,0,1,1,1})
        count = 1
        collisionNumber = collisionNumber + 1
    end

    simSetIntegerSignal("collFlag",colls)

    if(colls == 0 and count == 1) then
        simEndDialog(dialogHandle)
        count = 0
        colls = 0
    end
```

```
-------------------------------------------------------------------------------- Red Block-----------
---------------------------------

   if ((simCheckCollision(wheel_rr,blockR) ==1) or
(simCheckCollision(wheel_rl,blockR) ==1) or (simCheckCollision(wheel_fr,blockR)
==1) or (simCheckCollision(wheel_fl,blockR) ==1) or
(simCheckCollision(youBot,blockR) ==1)) then
      collsBlockR = 1
   else
      collsBlockR =0
   end



-----------------------------------------------------------------------------------------------------------
---------------------------------

-------------------------------------------------------------------------------- Blue Block----------
----------------------------------

   if ((simCheckCollision(wheel_rr,blockB) ==1) or
(simCheckCollision(wheel_rl,blockB) ==1) or (simCheckCollision(wheel_fr,blockB)
==1) or (simCheckCollision(wheel_fl,blockB) ==1) or
(simCheckCollision(youBot,blockB) ==1)) then
      collsBlockB = 1
   else
      collsBlockB =0
   end

--[[
   if (collsBlockB == 1 and   collsBlue == 0) then
   dialogHandle2 = simDisplayDialog('!!!!!!!              Good Job           !!!!!!! '," Move to
Next Block" ,sim_dlgstyle_ok,false,nil,{0,0.5,0,0,0,0},{0,0,0.2,1,1,1})
   collsBlue = 1
   end


   if( collsBlue == 1 and collsBlockB == 0 ) then
      simEndDialog(dialogHandle2)
      collsBlue  =  0
      collsBlockB = 0
    end

--]]
-----------------------------------------------------------------------------------------------------------
---------------------------------
```

```
---------------------------------------------------------------------------------- Green Block---------
-----------------------------------

   if ((simCheckCollision(wheel_rr,blockG) ==1) or
(simCheckCollision(wheel_rl,blockG) ==1) or (simCheckCollision(wheel_fr,blockG)
==1) or (simCheckCollision(wheel_fl,blockG) ==1) or
(simCheckCollision(youBot,blockG) ==1)) then
      collsBlockG = 1
   else
      collsBlockG =0
   end

--[[
   if (collsBlockG == 1 and   collsGreen == 0) then
   dialogHandle3 = simDisplayDialog('!!!!!!!            Good Job          !!!!!!! '," Move to
Next Block" ,sim_dlgstyle_ok,false,nil,{0,0.5,0,0,0,0},{0,0,0.2,1,1,1})
   collsGreen = 1
   end


   if( collsGreen == 1 and collsBlockG == 0 ) then
      simEndDialog(dialogHandle3)
      collsGreen  =  0
      collsBlockG = 0
    end

--]]
-----------------------------------------------------------------------------------------------------
---------------------------------


 if ((collsBlockR == 1 or collsBlockB == 1 or collsBlockG == 1)  and collisionR == 0)
then
      dialogHandle1 = simDisplayDialog('!!!!!!!            Good Job          !!!!!!! '," Move
to Next Block" ,sim_dlgstyle_ok,false,nil,{0,0.5,0,0,0,0},{0,0,0.2,1,1,1})
      collsRed = 1
      lastTime = currentTime
      collisionR  = 1
   end


   if( collsRed == 1 and  (currentTime - lastTime) > 4 ) then
      simEndDialog(dialogHandle1)
      collsRed  =  0
      collisionR  = 0
   end
   if ((simCheckCollision(wheel_rr,block1) ==1) or
(simCheckCollision(wheel_rl,block1) ==1) or (simCheckCollision(wheel_fr,block1)
==1) or (simCheckCollision(wheel_fl,block1) ==1) or
(simCheckCollision(youBot,block1) ==1)) then
       collsLaptop1 = 1
   else
       collsLaptop1 =0
   end

   if (collsLaptop1 == 1) then
   a2 = "  Round Complete.... Total Collision =  Tcoll   "
   b2 = string.gsub(a2, "Tcoll",collisionNumber-1)
   dialogHandle2 = simDisplayDialog('Game Over - Next Round
',b2 ,sim_dlgstyle_ok,false,nil,{0,0.5,0,0,0,0},{0,0,0.2,1,1,1})
   print(currentTime-timerStart)
   end


end
```

Modified Hukoyu Code

```
-- This is a ROS enabled Hokuyo_04LX_UG01 model (although it can be used as a
generic
-- ROS enabled laser scanner), based on the existing Hokuyo model. It performs
instantaneous
-- scans and publishes ROS Laserscan msgs, along with the sensor's tf.

-- Rajat Tyagi
--10/12/2016
-- This scripts converts the Hukoyu LIdar into a 360 LIDAR and
-- makes a rectangular forece field around the omnirobot ,  calculates the 3- DOF force-
feedback using the Fy Law and transmits the force-feedback to the joystick controller via
a ROS Node.

function round(num, idp)
  local mult = 10^(idp or 0)
  return math.floor(num * mult + 0.5) / mult
end


if (sim_call_type==sim_childscriptcall_initialization) then
   laserHandle=simGetObjectHandle("Hokuyo_URG_04LX_UG01_ROS_laser")
   jointHandle=simGetObjectHandle("Hokuyo_URG_04LX_UG01_ROS_joint")
   modelRef=simGetObjectHandle("Hokuyo_URG_04LX_UG01_ROS_ref")
   modelHandle=simGetObjectAssociatedWithScript(sim_handle_self)
   objName=simGetObjectName(modelHandle)


   scanR = 360
   stepN = 16
   maxDistance = 0.5
 --   compC = 0.1

   boundLayX = 0.15          -- Previous Value: 0.15
   boundLayY = 0.12        -- Previous Value: 0.15

   gainX = 250 * boundLayX      --30 -- 200
   gainY = 200 * boundLayY      --30 -- 200


gainQ = 2100 * boundLayX  --120
dgainQ = 100  * boundLayY   --100


   dgainX = 100 * boundLayX    --100
   dgainY = 750* boundLayY   --120

   ForceVecX = {}
   ForceVecY = {}
   momentVecX = {}
   momentVecY = {}

   mcm = 100

   feedbackString={}
   valsTransmit={}
   lastT = 0
   scanRange=scanR*math.pi/180 --You can change the scan range. Angle_min=-
scanRange/2, Angle_max=scanRange/2-stepSize
   stepSize=stepN*math.pi/1024
   pts=math.floor(scanRange/stepSize)
   --print(pts)
   dists={}
   ForceVec = {}
   points={}
   segments={}
   anglesT={}

   xVelocity = 0
   yVelocity = 0
   qVelocity = 0



   simSetObjectFloatParameter(laserHandle,sim_visionfloatparam_far_clipping,0.5)

   for i=1,pts*3,1 do
      table.insert(points,0)
   end
   for i=1,pts*7,1 do
      table.insert(segments,0)
   end

   black={0,0,0}
   red={0,0.6,0.9}
   redT={0,1,0}
   red1={1,0,0}
   green={0,1,0}
   blue={0,0,1}
   purple={0.5,0,0.5}
   gray={0.5,0.5,0.5}

   lines100=simAddDrawingObject(sim_drawing_lines,1,0,-
1,1000,black,black,black,red)
   lines1001=simAddDrawingObject(sim_drawing_lines,1,0,-
1,1000,black,black,black,redT)
   points100=simAddDrawingObject(sim_drawing_points,4,0,-
1,1000,black,black,black,red)
   linesFx=simAddDrawingObject(sim_drawing_lines,3,0,-
1,1000,black,black,black,green)
   linesFy=simAddDrawingObject(sim_drawing_lines,3,0,-
1,1000,black,black,black,blue)
   linesTq1=simAddDrawingObject(sim_drawing_lines,3,0,-
1,1000,black,black,black,red1)
   linesTq2=simAddDrawingObject(sim_drawing_lines,3,0,-
1,1000,black,black,black,purple)
   linesFr=simAddDrawingObject(sim_drawing_lines,3,0,-
1,1000,black,black,black,gray)

   pub=simExtRosInterface_advertise('/feedForce',  'std_msgs/String')
   pubData=simExtRosInterface_advertise('/data',  'std_msgs/String')
   pubDataTimed=simExtRosInterface_advertise('/dataT',  'std_msgs/String')

  -- pubX=simExtRosInterface_advertise('/feedForceX',  'std_msgs/Float64')
  -- pubY=simExtRosInterface_advertise('/feedForceY',  'std_msgs/Float64')
  -- pubQ=simExtRosInterface_advertise('/feedForceTheta',  'std_msgs/Float64')
```

```
end

if (sim_call_type==sim_childscriptcall_cleanup) then
   simRemoveDrawingObject(lines100)
   simRemoveDrawingObject(points100)
   simRemoveDrawingObject(linesFx)
   simRemoveDrawingObject(linesFy)
   feedbackString.data = tostring(0).. "," .. tostring(0)..",".. tostring(0)
   simExtRosInterface_publish(pub,feedbackString)


end

if (sim_call_type==sim_childscriptcall_sensing) then

showLaserPoints=simGetScriptSimulationParameter(sim_handle_self,'showLaserPoints')

showLaserSegments=simGetScriptSimulationParameter(sim_handle_self,'showLaserSeg
ments')

forceFeedback=simGetScriptSimulationParameter(sim_handle_self,'forceFeedbackParam
eter')
   ThetaFeedback=simGetScriptSimulationParameter(sim_handle_self,'ThetaFeedback')


   dists={}
   angle=-scanRange*0.5
   simSetJointPosition(jointHandle,angle)
   jointPos=angle

   laserOrigin=simGetObjectPosition(jointHandle,-1)
   laserOrient=simGetObjectOrientation(jointHandle,-1)
   modelInverseMatrix=simGetInvertedMatrix(simGetObjectMatrix(modelRef,-1))

countVec1 = 0

 ------------------------------------------------------------------


   for ind=0,12,1 do
      r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
      m=simGetObjectMatrix(laserHandle,-1)
      cos = math.cos(ind*stepSize)
      cosFac = (1-cos)/cos
      distX = boundLayX + 0.29
      fieldX = distX + (distX * cosFac)
      if r>0 and dist < fieldX then
```

```
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldX - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldX})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end

-----------------------------------------------------------------------
    for ind=13,31,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((90*math.pi/180)-ind*stepSize)
        cosFac = (1-cos)/cos
        distY = boundLayY + 0.19
        fieldY = distY + (distY * cosFac)

        if r>0 and dist < fieldY then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldY - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldY})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end

---------------------------------------------------------------------------

    for ind=32,51,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((-90*math.pi/180)+ind*stepSize)
        cosFac = (1-cos)/cos
        distY = boundLayY + 0.19
        fieldY = distY + (distY * cosFac)

        if r>0 and dist < fieldY then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldY - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldY})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end
------------------------------------------------------------------------
for ind=52,64,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((180*math.pi/180)-ind*stepSize)
        cosFac = (1-cos)/cos
        distX = boundLayX + 0.29
        fieldX = distX + (distX * cosFac)
        if r>0 and dist < fieldX then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldX - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldX})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end

------------------------------------------------------------------------------------------

for ind=65,76,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((180*math.pi/180)+ind*stepSize)
        cosFac = (1-cos)/cos
        distX = boundLayX + 0.29
        fieldX = distX + (distX * cosFac)
        if r>0 and dist < fieldX then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldX - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldX})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end

-------------------------------------------------------------------------------------------------------------
---------
for ind=77,95,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((270*math.pi/180)- ind*stepSize)
        cosFac = (1-cos)/cos
        distY = boundLayY + 0.19
        fieldY = distY + (distY * cosFac)

        if r>0 and dist < fieldY then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldY - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldY})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end

----------------------------------------------------------------------------------------------------

for ind=96,115,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos((-270*math.pi/180)+ ind*stepSize)
        cosFac = (1-cos)/cos
        distY = boundLayY + 0.19
        fieldY = distY + (distY * cosFac)

        if r>0 and dist < fieldY then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldY - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldY})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end
```

```
for ind=116,127,1 do
        r,dist,pt=simHandleProximitySensor(laserHandle) -- pt is relative to the laser ray!
(rotating!)
        m=simGetObjectMatrix(laserHandle,-1)
        cos = math.cos(-ind*stepSize)
        cosFac = (1-cos)/cos
        distX = boundLayX + 0.29
        fieldX = distX + (distX * cosFac)
        if r>0 and dist < fieldX then
            dists[ind]=dist
            ForceVec[ind] = mcm*(fieldX - dist)
            -- We put the RELATIVE coordinate of that point into the table that we will
return:
            ptAbsolute=simMultiplyVector(m,pt)
            ptRelative=simMultiplyVector(modelInverseMatrix,ptAbsolute)
            points[3*ind+1]=ptRelative[1]
            points[3*ind+2]=ptRelative[2]
            points[3*ind+3]=ptRelative[3]
            segments[7*ind+7]=1 -- indicates a valid point
        else
            dists[ind]=0
            ForceVec[ind] = 0
            countVec1 = countVec1 + 1
            -- If we didn't detect anything, we specify (0,0,0) for the coordinates:
            ptAbsolute=simMultiplyVector(m,{0,0,fieldX})
            points[3*ind+1]=0
            points[3*ind+2]=0
            points[3*ind+3]=0
            segments[7*ind+7]=0 -- indicates an invalid point
        end
        segments[7*ind+1]=laserOrigin[1]
        segments[7*ind+2]=laserOrigin[2]
        segments[7*ind+3]=laserOrigin[3]
        segments[7*ind+4]=ptAbsolute[1]
        segments[7*ind+5]=ptAbsolute[2]
        segments[7*ind+6]=ptAbsolute[3]
        anglesT[ind]= angle
        ind=ind+1
        angle=angle+stepSize
        jointPos=jointPos+stepSize
        simSetJointPosition(jointHandle,jointPos)
    end



    simAddDrawingObjectItem(lines100,nil)
    simAddDrawingObjectItem(lines1001,nil)
    simAddDrawingObjectItem(points100,nil)

    if (showLaserPoints or showLaserSegments) then
        t={0,0,0,0,0,0}
        for i=0,pts-1,1 do
            t[1]=segments[7*i+4]
            t[2]=segments[7*i+5]
            t[3]=segments[7*i+6]
            t[4]=segments[7*i+1]
            t[5]=segments[7*i+2]
            t[6]=segments[7*i+3]
            if showLaserSegments then
                simAddDrawingObjectItem(lines100,t)

            end
            if (showLaserPoints and segments[7*i+7]~=0)then
                simAddDrawingObjectItem(points100,t)
            end
        end
```

```
        for i=52,76,1 do
            t[1]=segments[7*i+4]
            t[2]=segments[7*i+5]
            t[3]=segments[7*i+6]
            t[4]=segments[7*i+1]
            t[5]=segments[7*i+2]
            t[6]=segments[7*i+3]
            if showLaserSegments then
              simAddDrawingObjectItem(lines1001,t)
            end
        end
    end



Fx = 0
Fy = 0
Tq = 0
Bx = 0
By = 0
Bq = 0
TFx = 0
TFy = 0
TFq = 0



    xVelocity = round(simGetFloatSignal("VeloX"),2)
    yVelocity = round(simGetFloatSignal("VeloY"),2)
    qVelocity = round(simGetFloatSignal("VeloQ"),2)
    TimeStamp = round(simGetFloatSignal("Time"),2)
    collFlag =   simGetIntegerSignal("collFlag")
    compC =     simGetFloatSignal("compC")


   hukoyuOrient = math.deg(laserOrient[3])



  if(forceFeedback) then
       for i=0,pts-1,1 do

       ForceVecX[i] = (ForceVec[i]*math.cos(-(scanRange/2)+((i-1)*stepSize)))
       ForceVecY[i] = (ForceVec[i]*math.sin(-(scanRange/2)+((i-1)*stepSize)))
       momentVecY[i] = (dists[i]*math.cos(-(scanRange/2)+((i-1)*stepSize))) - compC
```

```
        Fx = Fx + ForceVecX[i]
        Fy = Fy + ForceVecY[i]
        Tq = Tq + ForceVecY[i] * momentVecY[i]

        end

      if(countVec1<pts) then
       Fx= Fx/(pts-(countVec1))
       Fy= Fy/(pts-(countVec1))
       Tq =Tq/(pts-(countVec1))
      end


    Fx = -gainX * round(Fx,2)
    Fy = -gainY * round(Fy,2)
    Tq = gainQ * round(Tq,2)

     Bx = dgainX * xVelocity
     By = dgainY * yVelocity
     Bq = dgainQ * qVelocity

     TFx = Fx - Bx
     TFy = Fy - By


     if(ThetaFeedback) then

         TFq  = Tq - Bq

         else
         TFq = 0
         Tq = 0
         Bq = 0
     end


if (Fx ==0) then
     TFx = 0
end

if (Fy == 0) then
     TFy = 0
end

if (Tq == 0) then
     TFq = 0
```

```
end



 end

  feedbackString.data = tostring(TFx).. "," .. tostring(TFy)..",".. tostring(TFq)

    if ( (TimeStamp - lastT) > 0.5) then
         simExtRosInterface_publish(pubDataTimed,valsTransmit)

         lastT = TimeStamp
  end



simAddDrawingObjectItem(linesFx,nil)
simAddDrawingObjectItem(linesFy,nil)
simAddDrawingObjectItem(linesTq1,nil)
simAddDrawingObjectItem(linesTq2,nil)
simAddDrawingObjectItem(linesFr,nil)


function saturation(val)

 if (val > 255 ) then
 return 255
 end

 if (val < -255 ) then
 return -255
 end

 if (val >= -255 and val <= 255 ) then
 return val
 end
end


  ptForceX = {saturation(TFx)/100,0,0}

  m1=simGetObjectMatrix(modelRef,-1)
  ptRelFx=simMultiplyVector(m1,ptForceX)
  forceXcord =
{laserOrigin[1],laserOrigin[2],laserOrigin[3],ptRelFx[1],ptRelFx[2],ptRelFx[3]}

  simAddDrawingObjectItem(linesFx,forceXcord)
```

```
    ptForceY = {0,saturation(TFy)/100,0}
    ptRelFy=simMultiplyVector(m1,ptForceY)
    forceYcord =
{laserOrigin[1],laserOrigin[2],laserOrigin[3],ptRelFy[1],ptRelFy[2],ptRelFy[3]}

   simAddDrawingObjectItem(linesFy,forceYcord)


    ptForceR = {saturation(TFx)/100,saturation(TFy)/100,0}
    ptRelFr=simMultiplyVector(m1,ptForceR)
    forceRcord =
{laserOrigin[1],laserOrigin[2],laserOrigin[3],ptRelFr[1],ptRelFr[2],ptRelFr[3]}

    simAddDrawingObjectItem(linesFr,forceRcord)

    ptForceQ1 = {0,0,saturation(TFq)/200}
    ptRelTq1=simMultiplyVector(m1,ptForceQ1)

    if (TFq>0) then
       forceQ1cord =
{laserOrigin[1],laserOrigin[2],laserOrigin[3],ptRelTq1[1],ptRelTq1[2],ptRelTq1[3]}
       simAddDrawingObjectItem(linesTq1,forceQ1cord)
    end

    if (TFq<0) then
      forceQ1cord =
{laserOrigin[1],laserOrigin[2],laserOrigin[3],ptRelTq1[1],ptRelTq1[2],-ptRelTq1[3]}
      simAddDrawingObjectItem(linesTq2,forceQ1cord)
    end

   simExtRosInterface_publish(pub,feedbackString)
   simExtRosInterface_publish(pubData,valsTransmit)


end
```

REFERENCES

[1] ODV Industrial News. “Ground Support Worldwide Cover Story: Driving in Circles,” April 2010.

[2] Airtrax, "Sidewinder: Omni-directional Lift Truck, " unpublished.

[3] Kuka, “KMP Omnimove: Masterful maneuvering in confined spaces,” unpublished.

[4] O. Diegel, A. Badve, G. Bright, J. Potgieter, and S. Tlale, "Improved mecanum wheel design for omni-directional robots," *Australasian Conf. Robot. Autom., Auckland*, pp. 117-121, 2002.

[5] A. Gfrerrer, "Geometry and kinematics of the Mecanum wheel," *Comput. Aided Geometric Des.,* vol. 25, no. 9, pp. 784-791, 2008.

[6] S. A. Miller, "Network interfaces and fuzzy-logic control for a mecanum-wheeled omni-directional robot," unpublished.

[7] I. Omnix Technology, "Omnix Technology: Directional components and integrated syst.," unpublished.

[8] P. Viboonchaicheep, A. Shimada, and Y. Kosaka, "Position rectification control for Mecanum wheeled omni-directional vehicles," *Ind. Electron. Soc.,* vol. 1, pp. 854-859, 2003.

[9] A. Shimada, S. Yajima, P. Viboonchaicheep, and K. Samura, "Mecanum-wheel vehicle systems based on position corrective control," *Ind. Electron. Soc.,* vol. 6, pp. 2077-2082, 2005.

[10] A. Jochheim and C. Rohrig, "The virtual lab for teleoperated control of real experiments," in *Proc. 38th IEEE Conf. Decision and Control,* 1999, vol. 1, pp. 819-824.

[11] A. T. Bradley, S. A. Miller, G. A. Creary, N. A. Miller, M. D. Begley, and N. J. Misch, "Mobius, An Omnidirectional Robot Utilizing Mecanum Wheels and Fuzzy Logic Control," *Advances Astronautical Sci.,* vol. 121, pp. 251-266, 2005.

[12] G. Campion, G. Bastin, and B. Dandrea-Novel, "Structural properties and classification of kinematic and dynamic models of wheeled mobile robots," *IEEE Trans. Robot. Autom.,* vol. 12, no. 1, pp. 47-62, 1996.

[13] M. West and H. Asada, "Design of a holonomic omnidirectional vehicle," in *Robot. Autom., 1992 Proc. IEEE Int. Conf.*, pp. 97-103.

[14] S. A. Mascaro, "Force guided docking control of an omnidirectional holonomic vehicle and its application to wheelchairs," MIT, MA, 1997.

[15] S. Mascaro, J. Spano, and H. H. Asada, "A reconfigurable holonomic omnidirectional mobile bed with unified seating (RHOMBUS) for bedridden patients," in*Robotics and Automation, 1997 Proc. IEEE Int. Conf.*, vol. 2, pp. 1277-1282.

[16] Quaid III, Arthur E. "System and method for using a haptic device as an input device," U.S. Patent 8 095 200, Jan. 10, 2012.

[17] Rosenberg, Louis B. "Haptic feedback device," U.S. Patent 8 487 873, July 16, 2013.

[18] D. A. Grant and A. Kapelus, "Gaming device having a haptic-enabled trigger," U.S. Patent 814 258 644, Apr. 22, 2014.

[19] P. Bachman and A. Milecki, "MR haptic joystick in control of virtual servo drive," in *J. Physics, Conf. Series*, 2009, vol. 149, no. 1, p. 012034.

[20] R. J. Jacob, L. E. Sibert, D. C. McFarlane, and M. P. Mullen Jr, "Integrality and separability of input devices," IEEE/*ACM Trans. Computer-Human Interaction,* vol. 1, no. 1, pp. 3-26, 1994.

[21] A. Fattouh, M. Sahnoun, and G. Bourhis, "Force-feedback joystick control of a powered wheelchair: Preliminary study," in *Systems, Man and Cybernetics, 2004 IEEE Int. Conf.*, vol. 3, pp. 2640-2645.

[22] B. Woods and N. Watson, "A short history of powered wheelchairs," *Assistive Technol.,* vol. 15, no. 2, pp. 164-180, 2003.

[23] C. S. Harrison, M. Grant, and B. A. Conway, "Haptic interfaces for wheelchair navigation in the built environment," *Presence: Teleoperators Virtual Environments,* vol. 13, no. 5, pp. 520-534, 2004.

[24] B. Grychtol, H. Lakany, and B. A. Conway, "A virtual reality wheelchair driving simulator for use with a brain-computer interface," in *Postgraduate Conference in Biomedical Engineering & Medical Physics*, 2009, p. 67.

[25] C. E. Wong and A. M. Okamura, "The snaptic paddle: a modular haptic device," in *Eurohaptics Conf. Symp. Haptic Interfaces Virtual Environment and Teleoperator Syst.*, 2005, pp. 537-538.

[26] A. M. Okamura, "Methods for haptic feedback in teleoperated robot-assisted surgery," *Ind. Robot, Int. J.,* vol. 31, no. 6, pp. 499-508, 2004.

[27] R. B. Gillespie, M. Hoffinan, and J. Freudenberg, "Haptic interface for hands-on instruction in system dynamics and embedded control," in *Haptic Interfaces for Virtual*

Environment and Teleoperator Syst., 2003 Proc. 11th Symp., pp. 410-415.

[28] A. M. Okamura, C. Richard, and M. Cutkosky, "Feeling is believing: Using a force-feedback joystick to teach dynamic systems," *J. Eng. Edu.,* vol. 91, no. 3, pp. 345-349, 2002.

[29] M. Badescu, C. Wampler, and C. Mavroidis, "Rotary haptic knob for vehicular instrument controls," in *Haptic Interfaces for Virtual Environment and Teleoperator Syst., 2002 Proc. 10th Symp.*, pp. 342-343.

[30] D. Kim, K. W. Oh, D. Hong, J.-H. Park, and S.-H. Hong, "Remote control of excavator with designed haptic device," in *Control, Automation and Systems, 2008 Int. Conf.*, pp. 1830-1834.

[31] L. Marchal-Crespo, "Haptic guidance for enhancing human motor learning: Application to a robot-assisted powered wheelchair trainer," Citeseer, 2009.

[32] B. E. Dicianno, R. A. Cooper, and J. Coltellaro, "Joystick control for powered mobility: current state of technology and future directions," *Physical Med. Rehabil. Clinics North America,* vol. 21, no. 1, pp. 79-86, 2010.

[33] S. P. Levine, D. A. Bell, L. A. Jaros, R. C. Simpson, Y. Koren, and J. Borenstein, "The NavChair assistive wheelchair navigation system," *IEEE Trans. Rehabil. Eng.,* vol. 7, no. 4, pp. 443-451, 1999.

[34] B. Woods and N. Watson, "A short history of powered wheelchairs," *Assistive Technol.,* vol. 15, no. 2, pp. 164-180, 2003.

[35] R. C. Simpson, "Smart wheelchairs: A literature review," *J. Rehabil. Res. Develop.,* vol. 42, no. 4, p. 423, 2005.

[36] H. Niniss and T. Inoue, "Electric wheelchair simulator for rehabilitation of persons with motor disability," in *Symp. Virtual Reality VIII, Belém (PA)*, 2006.

[37] P. R. Giacobbi Jr, C. E. Levy, F. D. Dietrich, S. H. Winkler, M. D. Tillman, and J. W. Chow, "Wheelchair users' perceptions of and experiences with power assist wheels," *Amer. J. Physical Med. Rahabil.,* vol. 89, no. 3, pp. 225-234, 2010.

[38] C. Richard, A. M. Okamura, and M. R. Cutkosky, "Getting a feel for dynamics: Using haptic interface kits for teaching dynamics and controls," in *ASME 6th Annu. Symp. Haptic Interfaces, Dallas, TX, Nov*, 1997, pp. 15-21.

[39] K. Bowen and M. K. O'Malley, "Adaptation of haptic interfaces for a labview-based system dynamics course," in *Haptic Interfaces for Virtual Environment and Teleoperator Syst.*, 2006, pp. 147-152.

[40] D. E. Whitney, "Quasi-static assembly of compliantly supported rigid parts," *J. Dynamic Syst. Meas. Control,* vol. 104, no. 1, pp. 65-77, 1982.

[41] J. Urbano, K. Terashima, T. Miyoshi, and H. Kitagawa, "Impedance control for safety and comfortable navigation of an omni-directional mobile wheelchair," in *Intelligent Robots and Systems, Proc. IEEE/RSJ Int. Conf.*, 2004, vol. 2, pp. 1902-1907.

[42] Y. Kondo, T. Miyoshi, K. Terashima, and H. Kitagawa, "Navigation guidance control using haptic feedback for obstacle avoidance of omni-directional wheelchair," in *Haptic interfaces for virtual environment and teleoperator systems symp.,* 2008, pp. 437-444.

[43] Y. Ueno, H. Kitagawa, K. Kakihara, and K. Terashima, "Development of collision avoidance supporting system for power assist system in omni-directional mobile robot," in *SICE Annu. Conf. Proc.*, 2011, pp. 1447-1452.

[44] A. Fattouh, M. Sahnoun, and G. Bourhis, "Force-feedback joystick control of a powered wheelchair: Preliminary study," in *Systems, Man and Cybernetics, IEEE Int. Conf.*, 2004, vol. 3, pp. 2640-2645.

[45] L. Kitagawa, T. Kobayashi, T. Beppu, and K. Terashima, "Semi-autonomous obstacle avoidance of omnidirectional wheelchair by joystick impedance control," in *Intelligent Robots and Systems, 2001 Proc. IEEE/RSJ Int. Conf.*, vol. 4, pp. 2148-2153.

[46] Q. M. Christensen, *Three degree of freedom haptic feedback for assisted driving of holonomic omnidirectional wheelchairs*. The University of Utah, UT, 2011.

[47] Y. Kondo, T. Miyoshi, K. Terashima, and H. Kitagawa, "Navigation guidance control using haptic feedback for obstacle avoidance of omni-directional wheelchair," in *Haptic interfaces for virtual environment and teleoperator systems, symp.*, 2008, pp. 437-444.

[48] S. Robotics, "RMP 400 Omni: Segway Robotic Mobility Platforms," unpublished.

[49] P. Fiorini and Z. Shiller, "Motion planning in dynamic environments using velocity obstacles," *Int. J. Robotics Res.,* vol. 17, pp. 760-772, 1998.

[50] D. Bareiss, J. Van Den Berg, and K. K. Leang, "Stochastic automatic collision avoidance for tele-operated unmanned aerial vehicles," in *Intelligent Robots and Systems, 2015 IEEE/RSJ Int. Conf.*, pp. 4818-4825.

[51] S. Omari, P. Gohl, M. Burri, M. Achtelik, and R. Siegwart, "Visual industrial inspection using aerial robots," in *Applied Robotics for the Power Industry, 2014 3rd Int. Conf.*, pp. 1-5.

[52] Administration on Aging, "Aging Statistics," D. o. H. H. Services, unpublished.

[53] W. L. Erickson, C., "2007 Disability Status Report: United States," *Cornell University Rehabilitation Research and Training Center on Disability Demographics and Statistics,* unpublished.

[54] K. Berg, M. Hines, and S. Allen, "Wheelchair users at home: few home modifications

and many injurious falls," *Amer. J. Public Health,* vol. 92, no. 1, pp. 48-48, 2002.

[55] M. W. Brault, "Americans with Disabilities: 2005," *US Census Bureau,* unpublished.