UNIVERSITY OF CALIFORNIA,
IRVINE

Feedback Control for a Smart Wheelchair Trainer Based on the Kinect Sensor

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Mechanical and Aerospace Engineering

by

Aurelia McLaughlin Darling

Thesis Committee:
Professor David Reinkensmeyer, Chair
Professor Athanasios Sideris
Professor Faryar Jabbari

2014

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. David J. Reinkensmeyer, for providing an opportunity for me in his lab and all his great advice and motivation on my projects. Dr. Danny Zondervan, a fellow grad student in the lab, has also been instrumental in this research and has always been available when I needed help along the way, especially on the programming side. There would also be no project without Dr. Laura Marchal-Crespo, who started this work with her dissertation.

There are several professors who inspired me to acquire a Masters degree, Dr. Rida T. Farouki and Dr. Sanjay S. Joshi. If it were not for Dr. Joshi, I would not have had the love of controls that I do. If it were not for Dr. Farouki, I would never have applied to graduate school.

There are some unintentional heroes in this project as well, including several good friends, specifically Gabe and Mike, who have kept me sane and listened to my ideas throughout this process. I would not have gotten through this without my Mom, Jane McLaughlin, supporting me. I would also like to thank my good friends in Davis, California who are not always close to me, but are important nonetheless, Sonia, Shayna, Jenyne, Suzie, and Amanda. And especially thanks to my partner Andrew, without whom I would surely have destroyed several computers and various other hardware in my attempt to finish this thesis.

ABSTRACT OF THE THESIS

Feedback Control for a Smart Wheelchair Trainer based on the Kinect Sensor

BY:

Aurelia McLaughlin Darling

Master of Science in Mechanical and Aerospace Engineering

University of California, Irvine, 2014

Professor David Reinkensmeyer, Chair

This thesis describes a Microsoft Kinect-based feedback controller for a robot-assisted powered wheelchair trainer for children with a severe motor and/or cognitive disability. In one training mode, "computer gaming" mode, the wheelchair is allowed to rotate left and right while the children use a joystick to play video games shown on a screen in front of them. This enables them to learn the use of the joystick in a motivating environment, while experiencing the sensation and dynamics of turning in a safe setting. During initial pilot testing of the device, it was found that the wheelchair would creep forward while children were playing the games. This thesis presents a mathematical model of the wheelchair dynamics that explains the origin of the creep as a center of gravity offset from the wheel axis or a mismatch of the torques applied to the chair. Given these possible random perturbations, a feedback controller was developed to cancel these effects, correcting the system creep. The controller uses a Microsoft Kinect sensor to detect the distance to the screen displaying the computer game, as well as the left-right position (parallel parking concept) with respect to the screen, and then adjusts the wheel torque commands based on this measurement. We show through experimental testing that this controller effectively stops the creep. An added benefit of the feedback controller is that it approximates a washout filter, such as those used in aircraft simulators, to convey a more realistic sense of forward/backward motion during game play.

INTRODUCTION

Background and Significance

Independent mobility is essential for cognitive and social development in young children. Being mobile allows children to interact with others, explore their environment independently, and obtain a certain measure of autonomy[1,2,3]. For children with severe disabilities, this can be very difficult to achieve without the use of a powered wheelchair. However, due to the nature of their disabilities, many children find it difficult to learn to drive a powered wheelchair. Insurance companies will not pay for a powered wheelchair unless the recipient can demonstrate that they are able to handle one. These children are caught in a catch-22; they are unable to obtain a powered wheelchair because they do not have access to one they can practice with. Cognitive, behavioral and physical factors are the top three reasons a powered wheelchair is not recommended for a child when they are tested[4]. Funding, lack of family support, and transportation issues can be large factors contributing to why children do not receive a powered wheelchair when a health care provider recommended one.

Some groups have attempted to address this problem by developing smart wheelchairs that make driving easier. There are many examples of systems that attempt to create an environment that will better mobilize individuals and ultimately make mobility more realizable[5,6,7,8]. These generally require the user to give high level commands, such as verbalizing directions for chair movement, and then the chair would dispense with low level controls, such as avoiding collisions, maneuvering through doorways, or following others. They also allow for the ability to adjust the amount of control the user or system has at any given time. These systems do not encourage learning how to drive a powered wheelchair, instead opting to make driving more of an automated activity.

Other groups have proposed wheelchairs that can be used to help train children in the use of a powered wheelchair. This is different in that the children will be able to use a standard powered wheelchair instead of a specialized one that may be too expensive for them. An example of this is

described in one study that found two of three children were able to develop full control of a conventional powered wheelchair using safety devices put on the chair[9]. In another study, three of four children given training in powered wheelchairs gained independence in driving skills[10]. All families observed psychosocial improvements in their children most likely due to the increased independence, and one family is even requesting funding for a powered wheelchair.

Our lab chose this later approach to address the problem. The lab developed the Kinect-Wheelchair Interface Controlled (KWIC) smart wheelchair trainer, a wheelchair trainer platform that converts a manual wheelchair into a powered wheelchair[11,12,13]. This has the advantage of allowing the child to stay in their own seating system, while still being able to experiment and train with powered mobility. The KWIC has a video game mode which allows a child to learn in an engaging environment with minimal supervision. In this mode the forward/backward motion of the chair is turned off and the chair is only allowed to rotate in place. Games are presented on a screen in front of the chair that require the user to move the joystick left and right. There are three games: a balloon bursting game, a space invaders clone game, and a racing game. These range in difficulty: the balloon bursting game only requires left/right controls; the space invaders game requires left/right and forward controls; and the race car game requires left/right, and forward/backward controls[13]. As the user plays the games, they experience the consequences of the joystick motion, since the chair turns left and right.

In an initial usability study, the KWIC was loaned to a clinic for a year and therapists were able to use the device with eight children[13]. The therapists found the computer gaming mode useful in that it was very motivating for the children to practice. They also liked that it simplified control of the chair to left/right motions so this could be taught independently of forward/backward motions. However, during the usability study it was observed that the chair would slowly creep forward during game play. This slow forward motion is clearly undesirable as the child could collide with the wall if left

unsupervised during game play. The goal of this thesis was to identify the cause of the creep and to develop a sensor-based feedback controller for the chair to reduce or eliminate it.

METHODS

1. Equipment

The KWIC smart wheelchair trainer is shown from a front view in Figure 1 and from a back view in Figure 2. The rear wheels of a manual wheelchair can be rolled onto the cross-bars shown on the bottom of the trainer. The joystick is attached on an adjustable arm that is able to be repositioned as far as height relative to the ground and also able to swing outward and back in, so as to allow a wheelchair to be positioned on the trainer. There is a vertical metal tower in the center of the hardware above the child sitting in the chair that the Kinect and compass are attached to. As shown, a Microsoft Kinect is attached about a foot above the computer, while the compass is attached a few feet farther up from the Kinect.

There are four emergency stops that are effective for this machine. One is the emergency stop button, shown as the large red button on the surface behind the chair location. If you pull out any of the USB connections, the OMNI+, a low level controller that relays information from the joystick to the motors, will turn off, effectively shutting down the KWIC smart wheelchair trainer. Pressing the central computer spacebar also stops the program. There is one other stop, which is shutting down power to the motor, using a master kill switch.

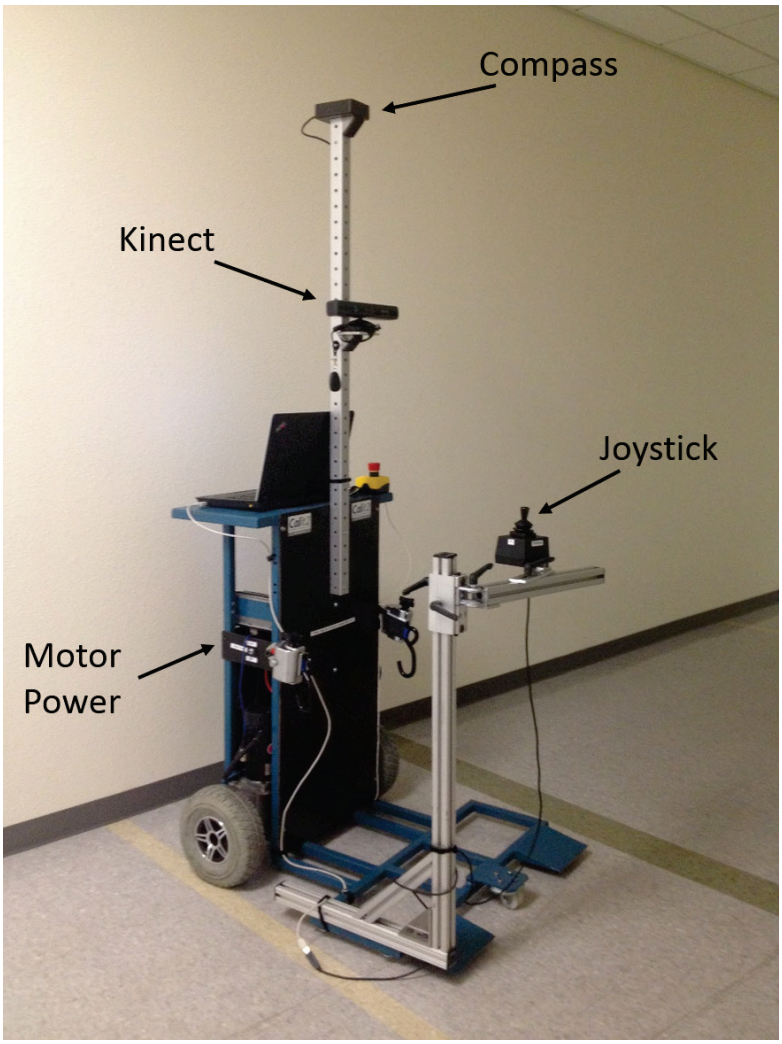


Figure 1: Front view of wheelchair trainer without a manual wheelchair mounted on it

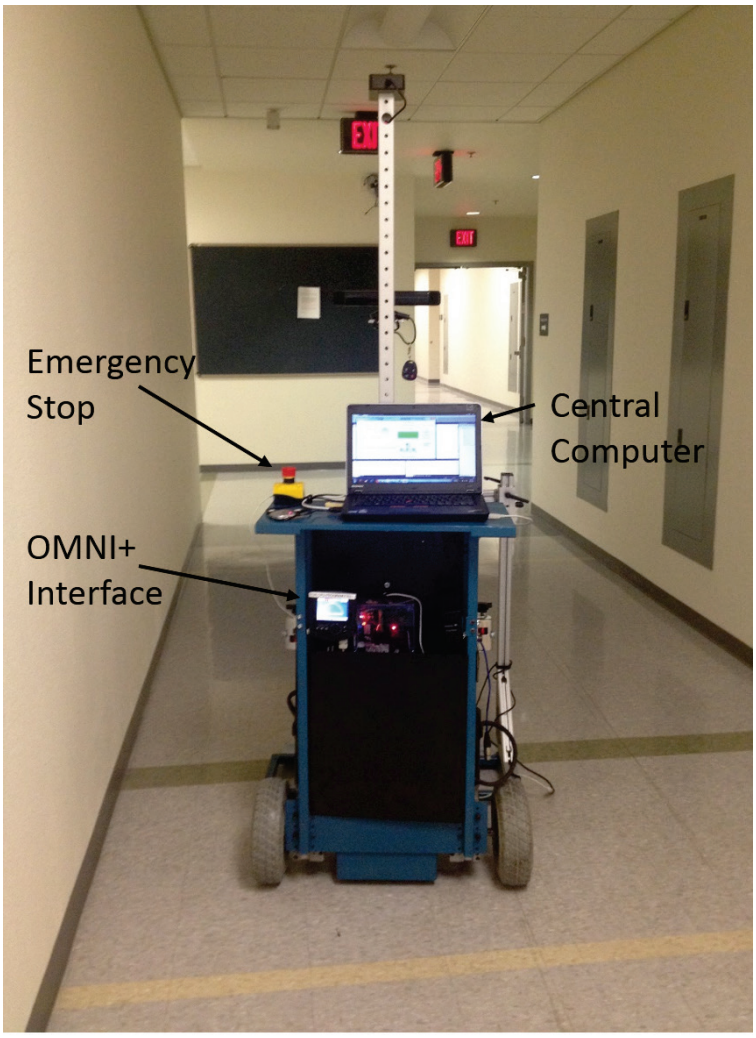


Figure 2: Back view of wheelchair trainer

2. Axes

Throughout this paper, I will refer to certain axes in relation to the KWIC Trainer and Kinect involved in this project. I define the Kinect coordinate frame as being positioned at the center of the Kinect camera. As shown in Figure 3, the Z axis is positive forward, the X axis is positive to the right, and the positive Y axis defines the height below the Kinect.

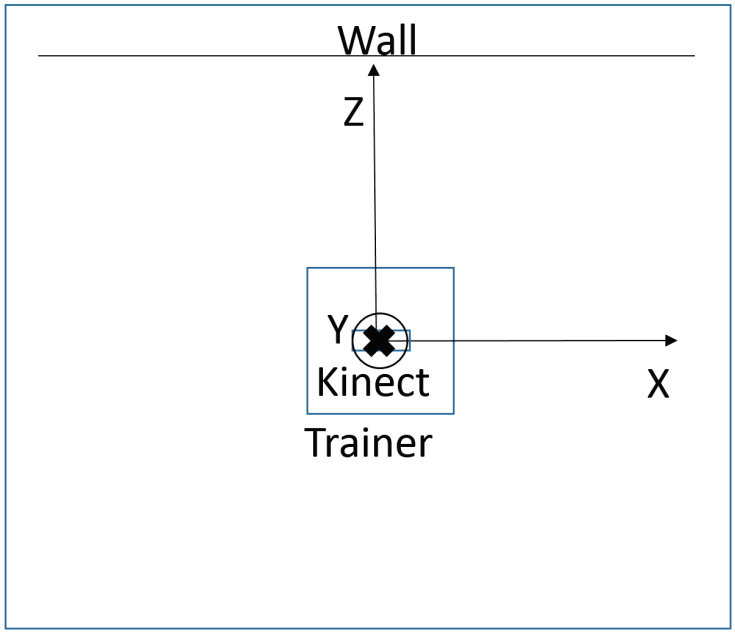


Figure 3: Axes orientation using Kinect as center

3. Model in Matlab

In order to better understand the dynamics of the KWIC trainer, a study was undertaken to model it. The model was implemented in Matlab, based on the model presented by Hu, Tiemin, et al.[14] Figure 4 below shows a simplified version of the robotic platform. The two driving wheels are shown as the back two and are on the same axis. The front wheel is separate from the back driving wheels and rotates freely. There are seven generalized coordinates that were used to describe the robot configuration:

$q = [x\ y\ \theta\ \phi_r\ \phi_l\ \dot{\phi}_r\ \dot{\phi}_l]^T$

As shown in Figure 4, the x and y are of the coordinates of O for this case. Theta is the orientation angle of the robot in terms of the robot frame. The variables ϕ_r, ϕ_l, $\dot{\phi}_r$ and $\dot{\phi}_l$, are the angles and angular velocities of the right and left driving wheels respectively. All the variables not explicitly stated are described in Table 1.

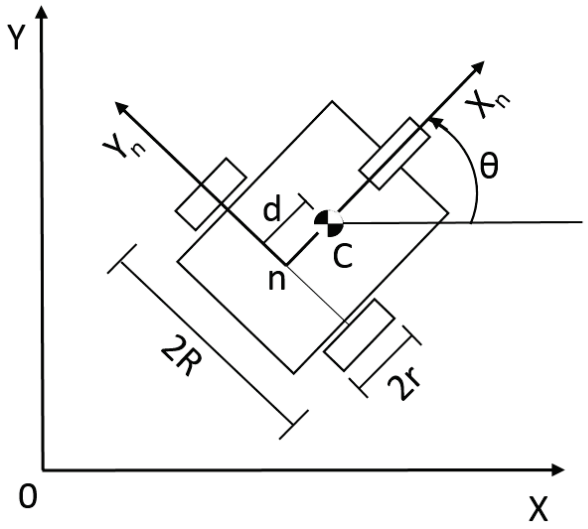


Figure 4: Mobile robot model

Table 1: Symbols used for mobile robot model

Symbol	Description
R	half of width of robot
r	radius of driving wheel
C	center of gravity
n	wheel axis origin
d	distance from n to C
θ	orientation angle of robot
ϕ_r	angle of right driving wheel
ϕ_l	angle of left driving wheel
τ_r	torque applied to right wheel
τ_l	torque applied to left wheel
m_c	mass of mobile robot platform
m_w	mass of one driving wheel with motor
m	total mass
I_c	MOI of platform about vertical axis through n
I_w	MOI of the wheel with the motor about wheel axis
I_m	MOI of the wheel with the motor about the wheel diameter

The dynamics of the wheelchiar are given by equation (1). In this equation M(q) is an n x n symmetric, positive matrix, C(q,$\dot{q}$) is the n x n centripetal coriolis matrix, E(q) is the n x r input transformation matrix,

τ is the r x 1 input vector $[\tau_r \tau_l]^T$, A(q) is the m x n Jacobian matrix associated with the constraints, and λ is the m x 1 constraints force vector.

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} = E(q)\tau - A^T(q)\lambda \quad (1)$$

The kinematic constraints can be espressed as equation (2):

$$A(q)\dot{q} = 0 \quad (2)$$

Let S(q) be an n-m full-rank matrix made up by a set of smooth and lienarly independent vectors spanning the null space of A(q):

$$A(q)S(q) = 0 \quad (3)$$

Using (2) and (3):

$$A(q)\dot{q} = 0^{3x1}$$

$$A(q)S(q) = 0^{3x2}$$

It then follows that the constrained velocity is always in the null space of A(q), and from (1) and (2) it is possible to find n-m volocities v(t) such that:

$$A(q)S(q)v(t) = 0^{3x2}\, v(t) = 0^{3x1}$$

$$A(q)\dot{q} = 0^{3x1} = A(q)S(q)v(t)$$

$$\dot{q} = S(q)v(t)$$

$$\ddot{q} = \dot{S}(q)v(t) + S(q)\dot{v}(t) \quad (4)$$

where v(t) is a vector defined as the angular velocities of the right and left wheels respectively,

$v(t) = [\dot{\phi}_r \; \dot{\phi}_l]^T$.

From substituting the equations found in (4) into (1) and multiplying by S^T, we find:

$$S^T(q)M(q)\,(\dot{S}(q)v(t) + S(q)\dot{v}(t)) + S^T(q)C(q,\dot{q})\,(S(q)v(t)) = S^T(q)E(q)\,\tau - S^T(q)A^T(q)\lambda$$

Since we already know that $S^T(q)A^T(q) = 0$, we can reduce the equation above to equation (5):

$$\bar{M}\dot{v} + \bar{C}v = \bar{B}\tau \quad (5)$$

Where:

$\bar{M}$ = S^TM S

$\bar{C}$ = S^T(M$\dot{S}$ + CS)

$\bar{B}$ = S^TE

Returning to the model in Figure 4, with the assumptions that the mobile robot is subjected to the conditions of pure rolling, non-slipping, and that it can only move in the direction normal to the driving wheels axis, the constraints are defined by the equations:

$\dot{y}\cos\theta - \dot{x}\sin\theta = 0$

$\dot{x}\cos\theta + \dot{y}\sin\theta + R\dot{\theta} = r\dot{\phi}_r$

$\dot{x}\cos\theta + \dot{y}\sin\theta - R\dot{\theta} = r\dot{\phi}_l$ (6)

This means that A(q) can be expressed as equation (7):

$$A(q) = \begin{bmatrix} sin\theta & -cos\theta & 0 & 0 & 0 \\ cos\theta & sin\theta & R & -r & 0 \\ cos\theta & sin\theta & -R & 0 & -r \end{bmatrix} \quad (7)$$

And from (2), we can find the S(q) matrix to be:

$$S(q) = \begin{bmatrix} \frac{r}{2}cos\theta & \frac{r}{2}cos\theta \\ \frac{r}{2}sin\theta & \frac{r}{2}sin\theta \\ \frac{r}{2R} & -\frac{r}{2R} \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

We can then find the $\bar{M}$, $\bar{C}$, and $\bar{B}$ matrixes to be:

$$\bar{M} = \begin{bmatrix} \frac{r^2(mR^2+I)}{4R^2} + I_w & \frac{r^2(mR^2-I)}{4R^2} \\ \frac{r^2(mR^2-I)}{4R^2} & \frac{r^2(mR^2+I)}{4R^2} + I_w \end{bmatrix}$$

$$\bar{C} = \begin{bmatrix} 0 & \frac{r^2 m_c d\dot{\theta}}{2R} \\ -\frac{r^2 m_c d\dot{\theta}}{2R} & 0 \end{bmatrix}$$

$$\bar{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

It should also be noted that $m = m_c + 2m_w$ and $I = m_c d^2 + 2m_w R^2 + I_c + 2I_m$.

Using these equations, a matlab function was written to find the angles and angular velocities of the two driving wheels given the initial position, orientation and velocity of the chair, and the wheel torques as a function of time:

```
function dx = Wheelchair1(t, x)
dx = zeros(7,1);

r= 5;                                       %rear wheel radius (inches)
R = 14;                                     %half of width of wheelchair(inches)
d = 0;                                      %distance from rear wheel axis and
                                            CG(inches)
mc = 100;                                   %mass of mobile robot platform (lbs)
mw = 50;                                    %mass of wheelchair with motor (lbs)
m = mc + 2*mw;                              %total mass (lbs)
% modeled as MOI of plate
Ic = (mc*((1/3) + ((36)^2/12)));            %MOI of platform about vertical axis
                                            through CG (lb-in^2)
% modeled as MOI of cylinder about z axis
Iw = (mw*r^2)/2;                            %MOI of wheel and actuator about
                                            wheel axis (lb-in^2)
% modeled as MOI of cylinder about x axis
Im = mw*(3*(r)^2+6^2)/12;
I = mc*d^2 + 2*mw*R^2 + Ic + 2*Im;          %total MOI (lb-in^2)
theta_dot = (r/(2*R))*(x(2) - x(4));        %angular velocity (radians/sec)
Tr = (cos((2*pi*t*.2))*450)+10;          %torque on right wheel axis(lb*in^2/s^2)
Tl = (-cos((2*pi*t*.2))*450);            %torque on left wheel axis(lb*in^2/s^2)

% Matrix Entries:
M1 = ((r^2*(m*R^2+I))/(4*R^2))+ Iw;
M2 = (r^2*(m*R^2-I))/(4*R^2);
M3 = (r^2*(m*R^2-I))/(4*R^2);
M4 = ((r^2*(m*R^2+I))/(4*R^2))+ Iw;
C1 = 0;
C2 = (r^2*mc*d*theta_dot)/(2*R);
C3 = -(r^2*mc*d*theta_dot)/(2*R);
C4 = 0;
B1 = 1;
B2 = 0;
B3 =0;
B4 = 1;

%Matrices
M = [M1 M2; M3 M4];
Ca = [C1 C2; C3 C4];
B = [B1 B2; B3 B4];
Phi_dot = [x(2); x(4)];
T = [Tr; Tl];
Phi_ddot = inv(M)*(B*T-Ca*Phi_dot);

%Simplify sine and cosine
```

```
C = cos(x(5));
S = sin(x(5));

%Equations
dx(1) = x(2);
dx(2) = Phi_ddot(1);
dx(3) = x(4);
dx(4) = Phi_ddot(2);
dx(5) = theta_dot;
dx(6) = ((r*x(4))+(R*theta_dot))/(C+(tan(x(5))*S));
dx(7) = ((r*x(2))-(R*theta_dot))/((cot(x(5))*C)+S);
```

An accurate model requires accurately estimating parameters of the model, such as the moment of inertia (MOI). A wheelchair is poorly modeled as a sphere or cube, but experimental methods that can be employed to find the moment of inertia are difficult to use with an electric powered wheelchair. For example, a common way of measuring moment of inertia of an object is by suspending it by one cable through its center of mass. One would then twist the object about said cable, measuring the period of oscillation in order to find the moment of inertia. This is not considered feasible for a wheelchair.

Generally an assumed mass moment of inertia is used or the wheelchair is simplified into regular cuboids[15]. However, one documented method[16] in finding this variable utilizes four cables and a flat disk to place the wheelchair on. The disk and wheelchair are suspended by the four cables and twisted slightly to find the moment of inertia of the wheelchair and disk combined. The moment of inertia of the disk is already known and can be subtracted from the total. I endeavored to use the moments of inertia found in this paper in the model above.

3.1 Results of Model

To simulate turning right and left repeatedly, we provided the simulated wheelchair with sinusoidal torques of amplitude 450 nm and frequency of 0.2 Hz on each wheel, with torques applied to each wheel 180 degrees out of phase. These equations for the torques are as follows:

```
Tr = cos(2*pi*t*.2)*450;
Tl = -cos(2*pi*t*.2)*450;
```

The initial conditions given are:

```
x_initial = [0, 0, 0, 0, -2*pi/180, 0, 0];
```

This is the initial angle of the right wheel, angular velocity of the right wheel, angle of the left wheel, angular velocity of the left wheel, angle of mobile robot, x position and y position. The mobile robot turns between -2 and +2 degrees for 30 seconds during the simulation.

Because the wheelchair trainer requires that the manual wheelchair be lifted up on the platform, the rear wheel axes of the trainer are not co- aligned with the rear wheel axes of the wheelchair (See Figure 1). Thus, the center of mass of the system, which is largely due to the weight of the driver, is forward from the platform rear wheel axes. When the distance between the center of gravity (CG) and the rear wheel axis (shown as d in the above code) is zero, the robot stays located at the same Z and X position (though the model is in x and y coordinates, this translates to Z and X coordinates in my axes system and is shown this way in the figures below for simplicity). However, when the center of gravity is moved forward from the rear wheel axes, i.e. when d is positive, indicating a center of gravity in front of the rear wheel axis, the simulations indicate that the robot moves forward as it turns repeatedly left and right “in place” (Figure 5). The robot starts out facing in the positive Z direction in the figure at the coordinate (0,0) and slowly starts moving forward in this position. Thus, one possible cause of the forward creep of the smart wheelchair trainer is that the center of mass of the chair is forward of the rear wheel axis.

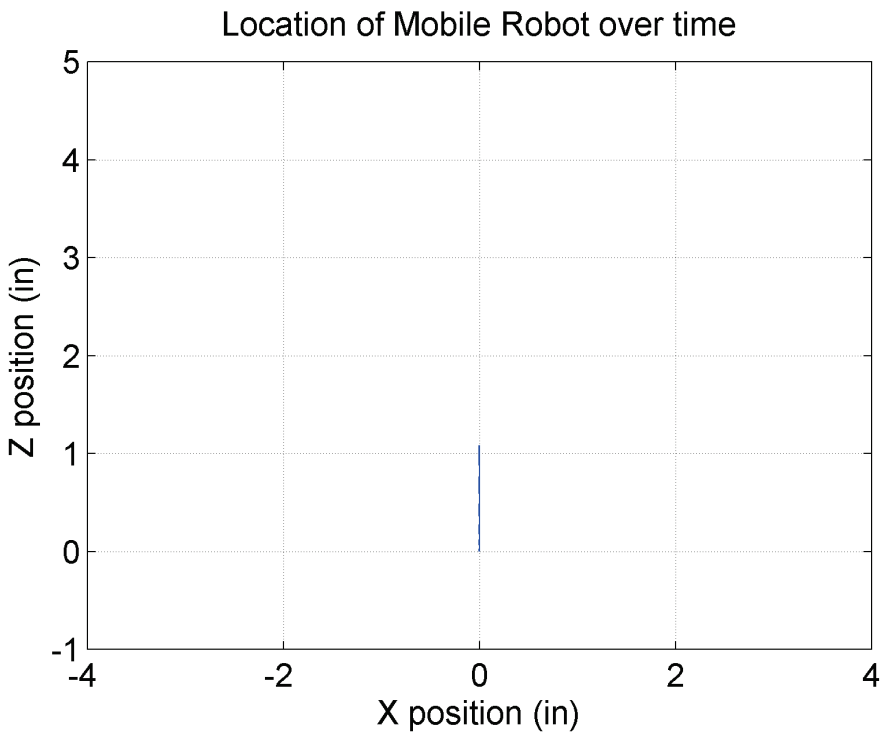


Figure 5: Location of mobile robot with CG displacement

Another possible cause of the wheelchair creep is a mismatch in the torques applied to each wheel. Returning the variable for distance between the center of gravity and the rear wheel axis to zero and adding a small perturbation to one of the torques applied in the form of offsetting the start position vertically of the cosine wave, yields the results shown in Figure 6. This small perturbation effectively works to move the robot forward and left over time in regards to its location. When the perturbation is applied to the opposite wheel, this yields the results shown in Figure 7, where the robot is moving forward and right over time. An example of the torque equations when the offset is in place on the right wheel is:

```
Tr = (cos((2*pi*t*.2))*450)+10;
Tl = (-cos((2*pi*t*.2))*450);
```

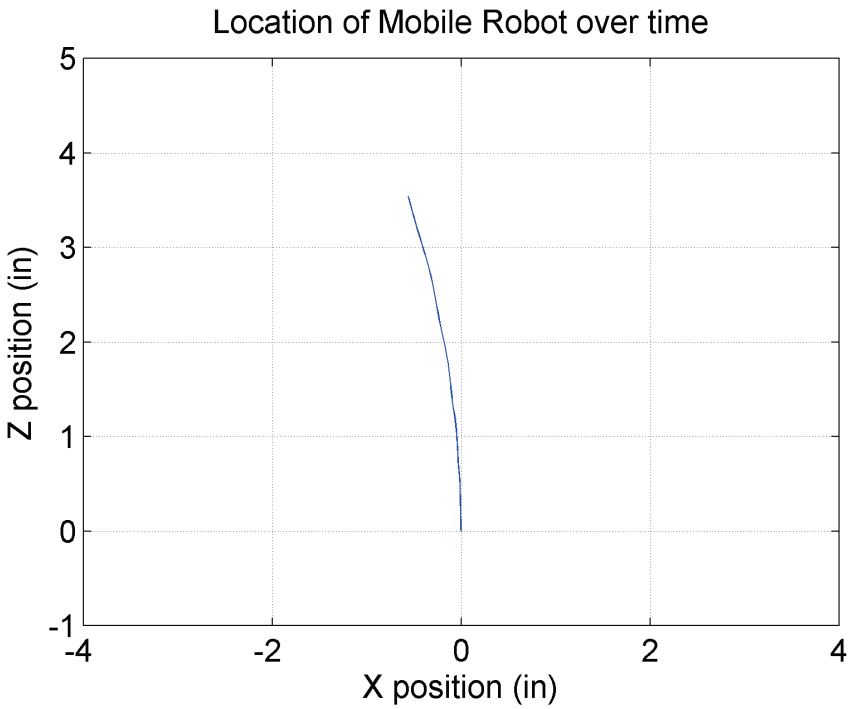


Figure 6: Location of mobile robot with torque perturbation of right wheel: The chair creeps forward as it turns repeatedly left and right over a 30 second period

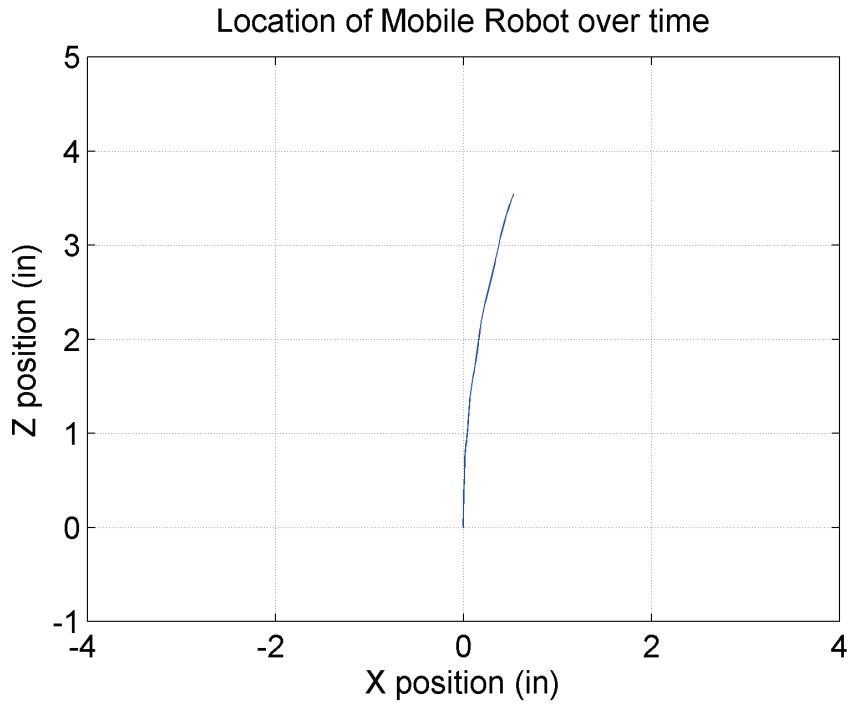


Figure 7: Location of mobile robot with torque perturbation of left wheel: The chair creeps forward as it turns repeatedly left and right over a 30 second period

Such a perturbation to the wheel torque could arise due to different calibrations of the wheel motors, different levels of initial friction in the wheel/motor assemblies, or different amounts of friction between the rear wheels and the floor, and thus can be viewed as a random perturbation. The remainder of this thesis focuses on developing a feedback controller to cancel the forward creep due to such a perturbation.

4. Finding Distance to the Wall

To develop a feedback controller for the chair, we required a measurement of the distance of the chair from the wall. We used the Microsoft Kinect Depth Camera for this purpose. The Microsoft Kinect was originally created for the Xbox system in November 2010 to be a motion sensing input device. There are three coordinate spaces the Kinect tracks; color, skeleton, and depth space[17]. The color space uses the color sensor to capture the color image of everything available in the field of view. The skeleton space utilizes the depth sensors to capture the 3D position of up to two human skeletons within range. The Depth space uses the depth sensors to capture a grayscale image of everything in the field of view. The depth sensors can optimally determine depth between 0.8 and 4 meters, or 31 inches and 157 inches.

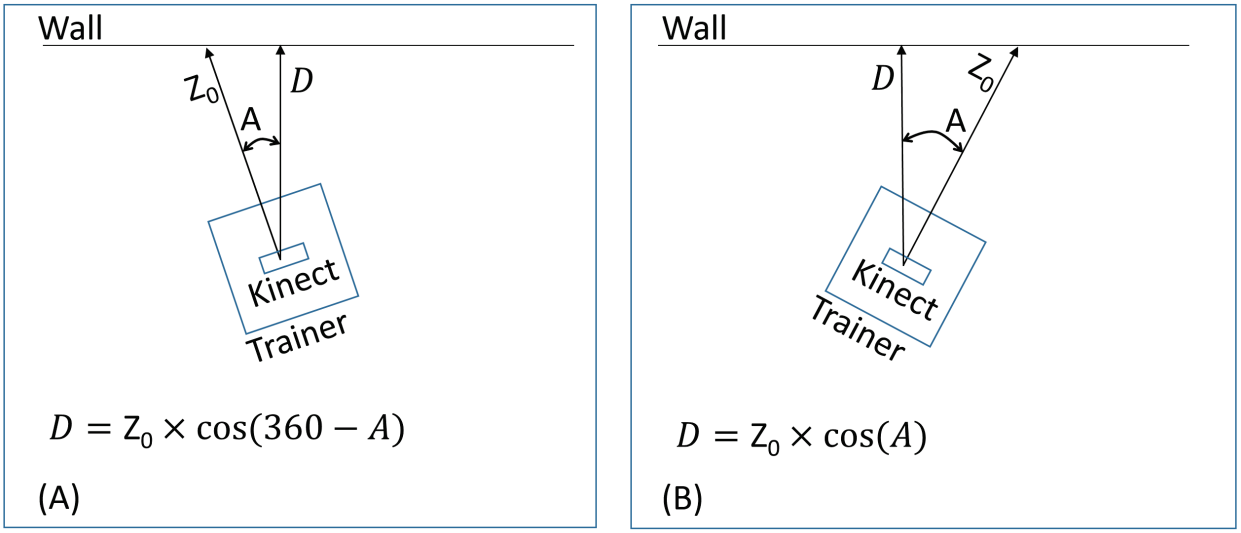
4.1 Case 1: Finding distance to the wall without a screen

Initially, I developed an algorithm to find the distance to the wall assuming there was no computer screen in front of the chair. This was because it was found that the Kinect depth camera would recognize the depth as zero for areas in the camera image directed at a computer or TV screen. The Kinect uses an infrared sensor to determine depth, and does this based on the amount of reflected infrared light from an object. If the surface of an object absorbs or redirects the light in a non-standard manner, such as glass for example, Kinect is unable to determine a depth and will return zero as the depth.

Figure 8 shows the kinematic model of the KWIC trainer and wall. In this case, D_l and D_r are defined differently because the compass value jumped from 0 to 360 degrees when the chair rotated right. This was taken into account in a later control algorithm. Thus, while the trainer is turned to the left in relation to the wall (Figure 8A), the angle is read from 3600 and goes down. Conversely, as the trainer turns right in relation to the wall (Figure 8A), the angle starts from 0 and goes up. As shown in Figure 8, we can find that:

$D = Z_0 \times \cos(360 - A)$

$D = Z_0 \times \cos(A)$



Trainer Oriented Left in Relation to Wall Trainer Oriented Right in Relation to Wall

Figure 8: Kinematic model of trainer and wall: (A) oriented to left; (B) oriented to right

Note that the distance to the wall is related to the Z distance from the Kinect by a cosine function, because the Kinect is firmly attached to the trainer and moves with it.

The Kinect returns a 320x240 array of depth readings that must be analyzed to find Z. Note that when facing the wall, the Kinect views the right most pixel as pixel 1 and the left most pixel as pixel 320. I ensemble averaged the first five rows of pixels to smooth the data, forming an 320x1 array, AvePix. I chose the first 5 rows instead of all 240 rows, because sometimes the chair user's body would affect the depth readings of lower rows. Using this array and the compass angle, I can then find the distance to

the wall. I use the center pixel in the depth camera image to measure Z_0, then the equations in Figure 8 to calculate the depth D, given the compass angle A. The following is code that determines the Z and X distances from the Kinect. AvePix is the array that stores the Z distances from the Kinect, which it does with the function depthPixels[].Depth.

The X distance is also found in this piece of code. The array that holds all of the X position values is called XPos. In order to find the X distance, several variables had to be defined first. One was the width of the screen, being 320 pixels. The height of the field of view also was needed, in this case 58.5 degrees. This number varies depending on the resolution. A variable that represents width over distance is then created, or X over z. This variable is non-dimensionalized, so that later we can simply multiply this variable by the actual Z distance, which has dimensions in millimeters, to find the X distance. The origin of the axes is from the Kinect, though I defined both directions for X as positive values for easier calculations later.

The following code is used to find X and Z distance arrays:

```
double[] AvePix = new double[320]; //array to store average pixel depth across screen, 5
down
double[] XPos = new double[320]; //array for x position of pixel

int a = 0; // stores the total depth value for array
int c = 0; //stores distance out (x direction) in array XPos
int numLines = 5; //number of lines being used to store the depth data and x position
data
int horizontalRes = 320; //horizontal resoltuion of the camera

//variables used to find x distancce
double depthWidth = 320;
double depthWidthHalf = depthWidth / 2;
double depthHFOV = 58.5;  //height field of view(different depending on number of pixels)
double depthH = 2*(Math.Tan((depthHFOV / 2) * (Math.PI / 180))); //this is equal to width
over distance
//to just get width, you must multiply by the distance, z in this case, which is done
below
double mmtoin = .03937;  //used to change mm to inches, since kinect gives you mm data

int increment = 0;

for (int i = 0; i < this.depthPixels.Length; ++i)
{
  // Get the depth for this pixel
  short depth = depthPixels[i].Depth;
```

```
  if (increment < horizontalRes)
  {
    if (i == increment)
    {
      short depth1 = depthPixels[i].Depth;
      AvePix[a] = AvePix[a] + depth1;
      //What follows is the math required to shift the origin to the center of the kinect
depth camera
      //and find the x distance proportionate to the depth data of that pixel
      if (c >= 0 && c <= 159)
      {
        XPos[c+(160-c-(c-1))] = Math.Round((((depth1 * depthH)/2) * ((c + 1) /
depthWidthHalf)) * mmtoin);
      }
      if (c >= 160)
      {
        XPos[c] = Math.Round(((depth1 * depthH/2) * ((c - 159) / depthWidthHalf)) *
mmtoin);
      }
      increment = ++increment;
      a = ++a;
      c = ++c;
      //start it over again when it reaches the end of the line, but not more than 5
lines
      if (increment == horizontalRes && horizontalRes < 1600)
      {
        c = 0;
        a = 0;
        horizontalRes = horizontalRes + 320;
      }
    }

  }
}
```

There were several safeguards put in place, such as if the Kinect found a distance from the wall that was either too close to be accurate, or a distance too far outside of the Kinect's operating range, then the Z distance reverted to the desired Z distance in order to ensure the trainer would not move on incorrect information. In this case, if it is closer than 4 feet or farther away than 11.5 feet, the control algorithm will not work. It was also determined that the compass is more accurate than calculating the angle every time to find the distance to the wall, so the move was made to exclusively use the compass for the angle.

4.2 Case 2: Finding distance to the wall when a screen is present

To determine the true distance to the wall when a computer screen is in front of the chair, as is the case during normal use of the smart wheelchair trainer in "computer gaming mode", the effect of the screen on the depth reading must be taken into account. An example of what the Kinect calculates with and without the screen present is shown in Figure 9. With just the wall present, the Kinect calculates the Z and X distances without any interruptions. When the screen in present though, there is a section of about 27 inches (the current width of the screen) that is read as zero for both Z and X distances. The X values for this section are read as zero because they are calculated using the Z distances.

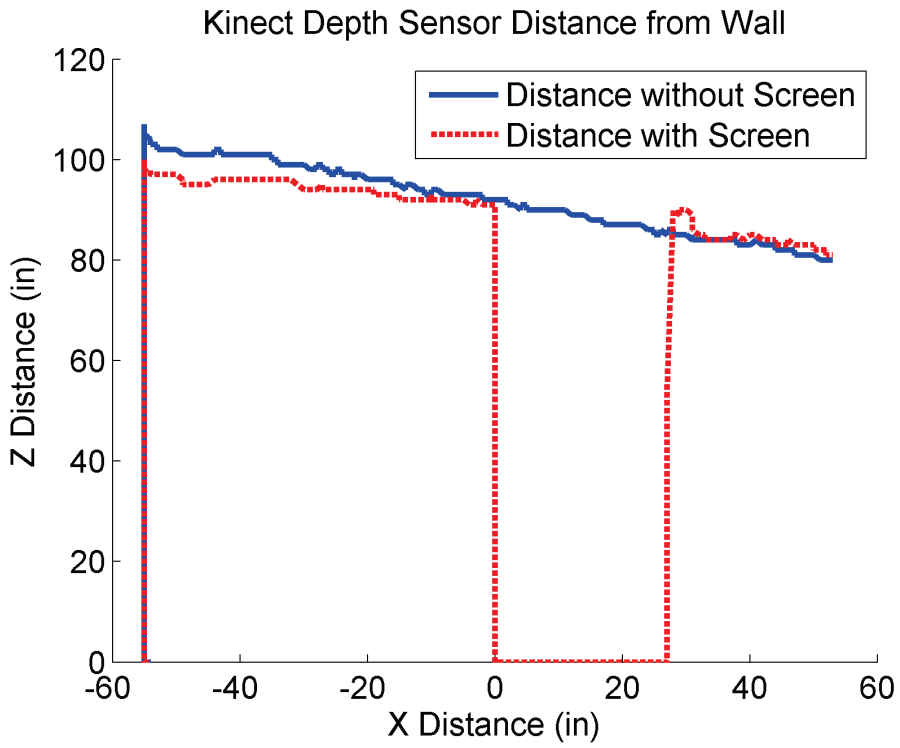


Figure 9: Kinect depth sensor distances with and without a screen in the field of view

In order for this algorithm to work properly with a TV screen, I first search for a pixel with a non-zero depth reading, starting from the center pixel and moving right. Once this pixel is found, an extra correction value is needed, as shown in Figure 10. D is found by first calculating D_a and then adding or

subtracting the correction factor, D_{al} or D_{ar}. Z_a is exaggerated in order to show the process. We find that the desired distances are as follows:

$$D = D_a + D_{al} = Z_a \times \cos(A - 360) - X_{al} \times \sin(A - 360)$$

$$D = D_a - D_{ar} = Z_a \times \cos(A) - X_{ar} \times \sin(A)$$

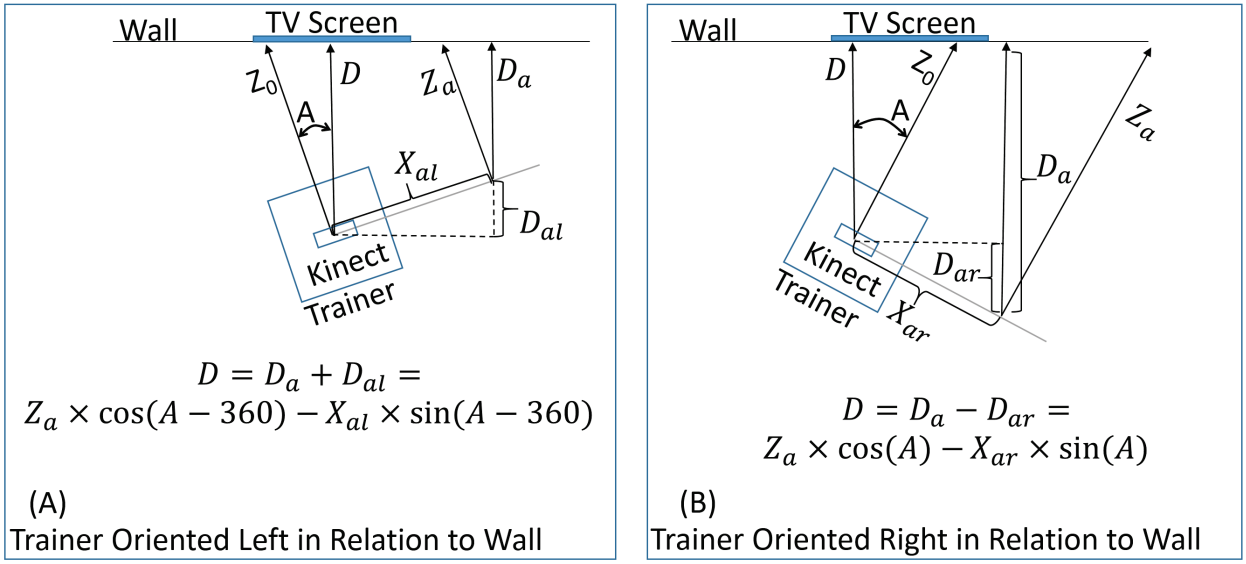


Figure 10: Distance to wall when screen present: (A) oriented to left; (B) oriented to right

Note that the extra D values are both subtracted in these formulas. Recall that cosine is an even function and sine is an odd function. Therefore, the negative angle found when the trainer is turned left comes into play only for the sine functions, making the second term in the D_l equation a negative value. This code executes the calculation:

```
for (int pix = xpix; pix >= 50; pix--)
{
   if (AvePix[pix] != 0 && XPos[pix] != 0)
   {
      if (heading >= 0 && heading < 60)
      {
         WallAng = heading;
         double headingRad = WallAng * (Math.PI / 180);
         AddedXVal = Math.Sin(headingRad) * XPos[pix - 2];
         if (AddedXVal > 12 || AddedXVal < 0)
         {
            AddedXVal = 0;
         }
         WallDist = (AvePix[pix - 2] * Math.Cos(headingRad)) - AddedXVal;
         break;
      }
```

```
        if (heading <= 360 && heading > 300)
        {
            WallAng = heading - 360;
            double headingRad = WallAng * (Math.PI / 180);
            AddedXVal = Math.Sin(headingRad) * XPos[pix];
            if (AddedXVal < -12 || AddedXVal > 0)
            {
                AddedXVal = 0;
            }
            WallDist = (AvePix[pix] * Math.Cos(headingRad)) - AddedXVal;
            break;
        }

    }
}
```

In this algorithm we have a predetermined center pixel value of 160. The program will decrease in pixel value, from the center pixel until it finds one that is associated with a positive depth reading for both the X and Z values. Once this pixel is found, the program determines if the trainer is between 0 and 60 degrees or 360 and 300 degrees. If the angle is the latter, 360 is subtracted from it, making it negative. An extra D distance is calculated, which would be positive if turned right and negative if turned left. As long as this value is within an acceptable range, it is subtracted (or added for D_l) from the initial D distance. This is outlined in the flow chart below.

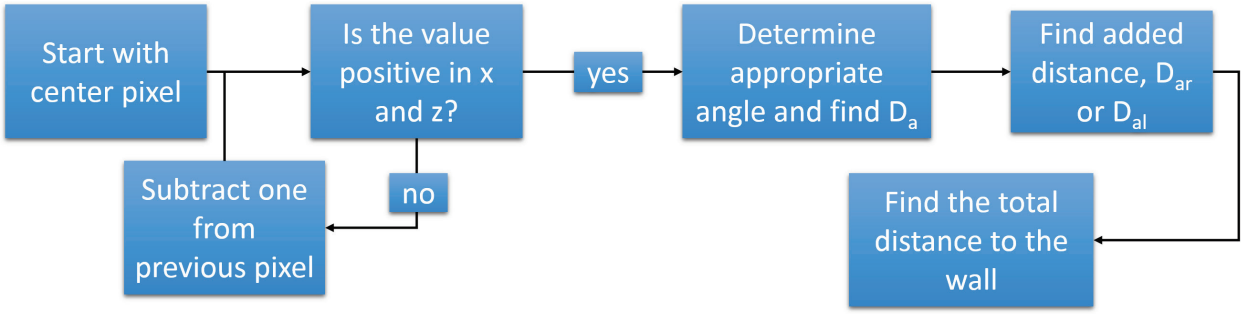


Figure 11: Flow chart of finding distance to wall

4.3 Depth Controller

Using the distance to the wall calculated as above, a controller can be used to change said distance in relation to a desired distance. This was implemented using a proportional controller:

```
DeltaY = GainYint * ((int)WallDist - DesWallDist);
```

DeltaY in this case is a value added to motor torque in the motor control section of the code where Y is defined as the axis on which the trainer moves forward and backward. The user sets the desired distance from the wall, DesWallDist, in a box in the user interface. The program makes sure the trainer is within 5 degrees from 0 and that an appropriate distance has been set, then it sets the desired distance to that entered. There is a separate button to turn this section of the code on by setting GainYint to 5, which will create a non-zero DeltaY value, which is used as follows:

```
controlValueY = controlValueY + ((double)DeltaY/100.0);
```

This will effectively change the value given to the motors for the Y control value. When the trainer is closer to the wall than the desired distance, the Y control value will be less than without the Delta Y value and vice versa. This will move the trainer backward when it is closer to the wall than desired and move the trainer forward when it is farther from the wall than desired.

5. Estimating X Position of KWIC Trainer

The fact that the Kinect reads zero as the distance to the screen we are working with, was an opportunity to use this as a means to find the X translation (i.e. left/right translation) of the smart wheelchair trainer over time. We can track the position of the screen in relation to the trainer over time, creating a way to track the X position. A way to do this is shown in Figures 12 and 13. In Figure 12, the trainer is oriented right in relation to the wall and we are determining the distance from the center of the Kinect to the right side of the screen, X_{tr}. In Figure 12A, we determine X_{tr} by determining X_1 and X_2 and summing them. In Figure 12B, X_{tr} is found by determining D_2 and using the angle found by the compass. Figure 13 follows the same logic, but the trainer is turned to the left instead. The equations found shown in Figures 12 and 13 are:

$$X_{tr} = X_1 + X_2 = D \times \tan(A) + X_r / \cos(A)$$

$$X_{tr} = D_2 \times \tan(A) = (D - X_r / \sin(A)\,) \times \tan(A)$$

$$X_{tl} = X_1 + X_2 = D \times \tan(360 - A) + X_l / \cos(360 - A)$$

$$X_{tl} = D_2 \times \tan(360 - A) = (D - X_l / \sin(360 - A)) \times \tan(360 - A)$$

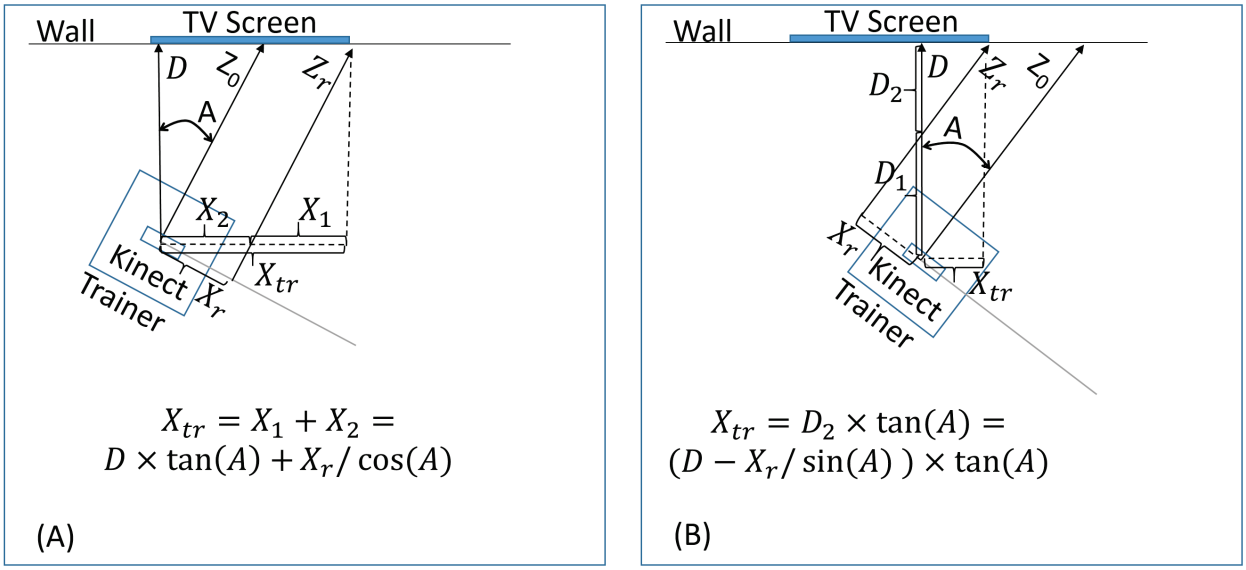


Figure 12: Model showing X position from Kinect to right side of screen: (A) X_{tr} determined by summing X_1 and X_2; (B) X_{tr} determined by using D_2 and Compass

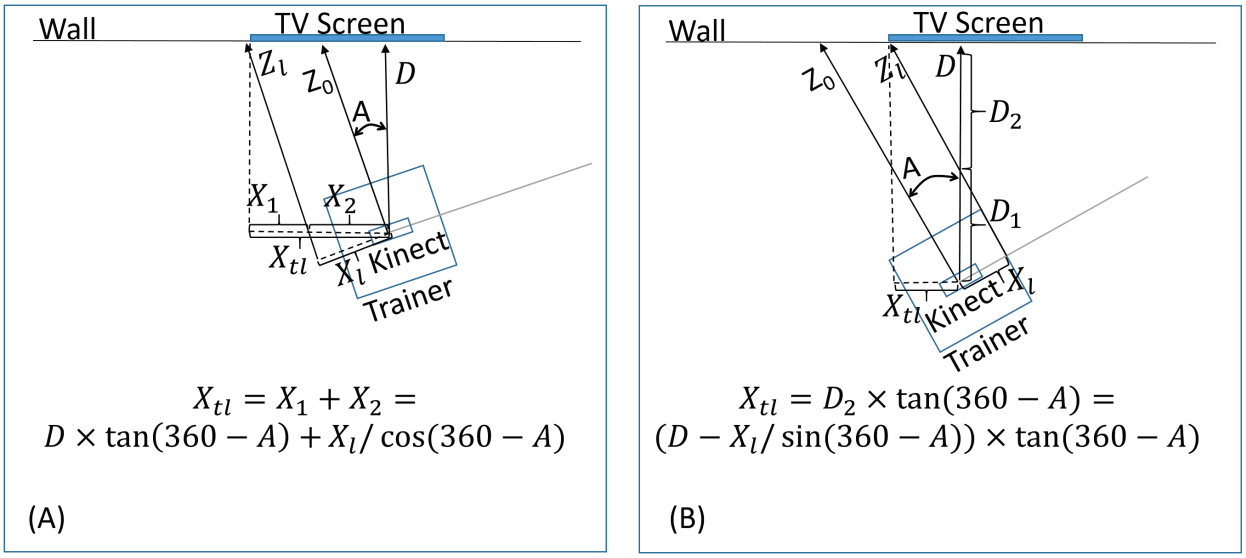


Figure13: Model showing X position from Kinect to left side of screen: (A) X_{tl} determined by summing X_1 and X_2; (B) X_{tl} determined by using D_2 and Compass

In these equations we are dealing with several trigonometric functions, so finding the appropriately signed angle is critical. This is why the angle when turned to the left must be subtracted from 360 and not the other way around, as shown in previous equations.

The code to execute this is as follows:

```
//to find the distance translated in the x position
   for (int pixinx = 10; pixinx <= 310; pixinx++)
   {
      if (AvePix[pixinx] == 0)
      {
         xtranslationR = pixinx;
         RSideScreenInches = (int)XPos[pixinx-2];
         RSideZDist = AvePix[pixinx - 2];
         break;
      }
   }

   for (int pixinxL = 310; pixinxL >= 10; pixinxL--)
   {
      if (AvePix[pixinxL] == 0)
      {
         xtranslationL = pixinxL;
         LSideScreenInches = (int)XPos[pixinxL+2];
         LSideZDist = AvePix[pixinxL + 2];
         break;
      }
   }
```

In the above section, we search for the first pixel from the right and left of the Kinect view frame that equals zero. When that pixel is found, it determines the X and Z distances to that pixel location for both the right and left side of the screen. From the previous figures, this would be X_r, Z_r, X_l, and Z_l, respectively. In this way, it finds the distance from the center of the Kinect to the edge of the screen on the right and left as well as the distance to the wall at those points. Note that for the pixel found on the right, two pixel values are subtracted from that pixel to find the X and Z distances, and two pixel values are added to the pixel found for the left side of the screen. Otherwise, the values would read zero. A flow chart of this is shown in Figure 14.

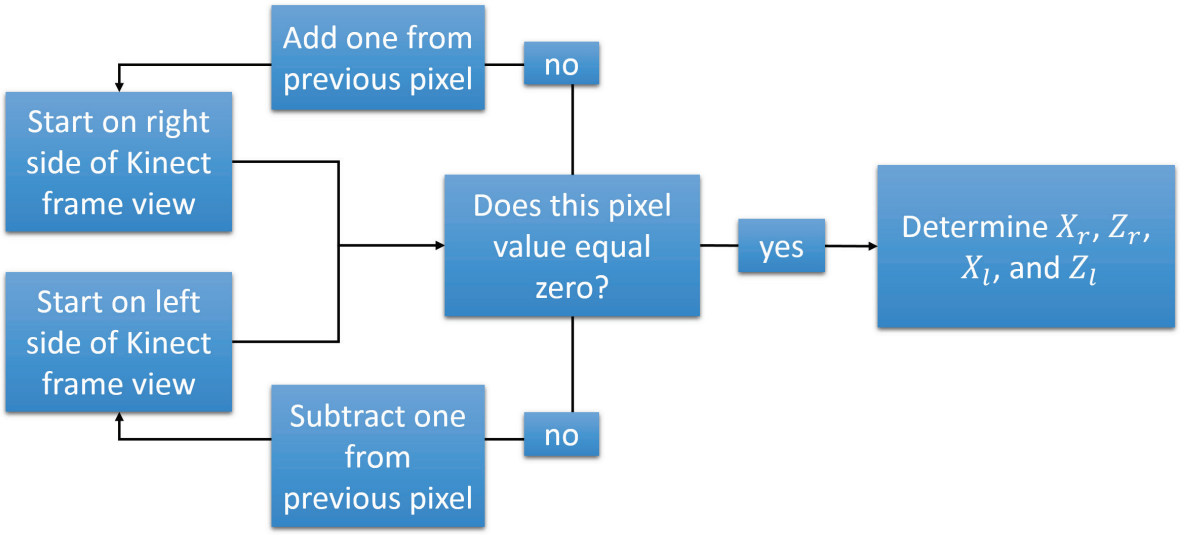


Figure 14: Flow chart to determine X_r, Z_r, X_l, and Z_l

The next section of code is as follows:

```
if (heading > 0 && heading < 60)
{
   HeadingRad = heading * (Math.PI / 180);
   if (xtranslationR > 160)
   {
      z1 = RSideScreenInches / Math.Sin(HeadingRad);
      z2 = WallDist - z1;
      RightX = (z2) * Math.Tan(HeadingRad);
   }
   if (xtranslationR <= 160)
   {
      x1 = (WallDist * Math.Tan(HeadingRad));
      x2 = (RSideScreenInches / Math.Cos(HeadingRad));
      RightX = x1 + x2;
   }
   ScreenSizeInches = (int)RightX + (int)LeftX;
}
if (heading > 300 && heading < 360)
{
   HeadingRad = (360 - heading) * (Math.PI / 180);
   if (xtranslationL < 160)
   {
      z1 = LSideScreenInches / Math.Sin(HeadingRad);
      z2 = WallDist - z1;
      LeftX = (z2) * Math.Tan(HeadingRad);
   }
   if (xtranslationL >= 160)
   {
      x1 = WallDist * Math.Tan(HeadingRad);
      x2 = LSideScreenInches / Math.Cos(HeadingRad);
      LeftX = x1 + x2;
   }
   ScreenSizeInches = (int)RightX + (int)LeftX;
}
```

```
double XTranslation = 0;
if (ScreenSizeInches >= 23 && ScreenSizeInches <= 29)
{
   ScreenDifference = (int)LeftX - (int)RightX;
   double XTranslation = ScreenDifference/2;

}
```

In this section of the code, the equations to find the true distances from the center of the Kinect to the right and left sides of the screen, X_{tr} and X_{tl}, are found. Then the width of the screen, ScreenSizeInches in the code, is determined by adding these two values. As long as this variable is within an acceptable range, the difference between the right and left side is found. The actual X translation is this value divided by two, XTranslation in the code. This difference is the way we are able to track the movement of the trainer over time. A flow chart of this is outlined in Figure 15.

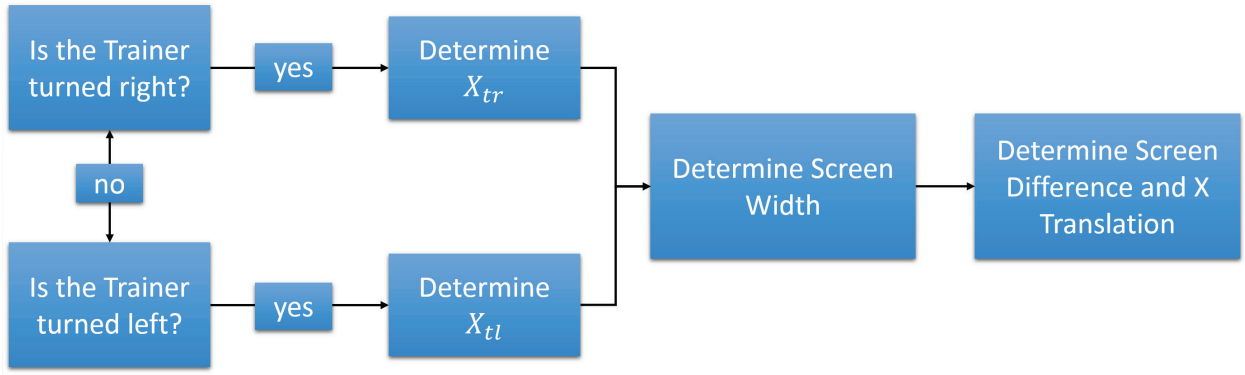


Figure 15: Flow chart to determine X translation

5.1 Lateral Controller

Due to the system being nonholonomic, there was no command to make the KWIC Trainer move sideways; it only moved forward and backward, and turned right and left. I knew from pilot testing that the trainer will move sideways over the course of turning, a parallel parking effect, under certain circumstances. For example, when I had the control algorithm on in preliminary testing, I found that it moved to the left while staying at a predetermined distance from the wall (Figure 16). This would occur due to the trainer estimating itself too close to the wall while turned right and too far from the wall

when turned left. This caused it to repeatedly move backward while facing right and forward while facing left, causing the parallel parking effect to the left.

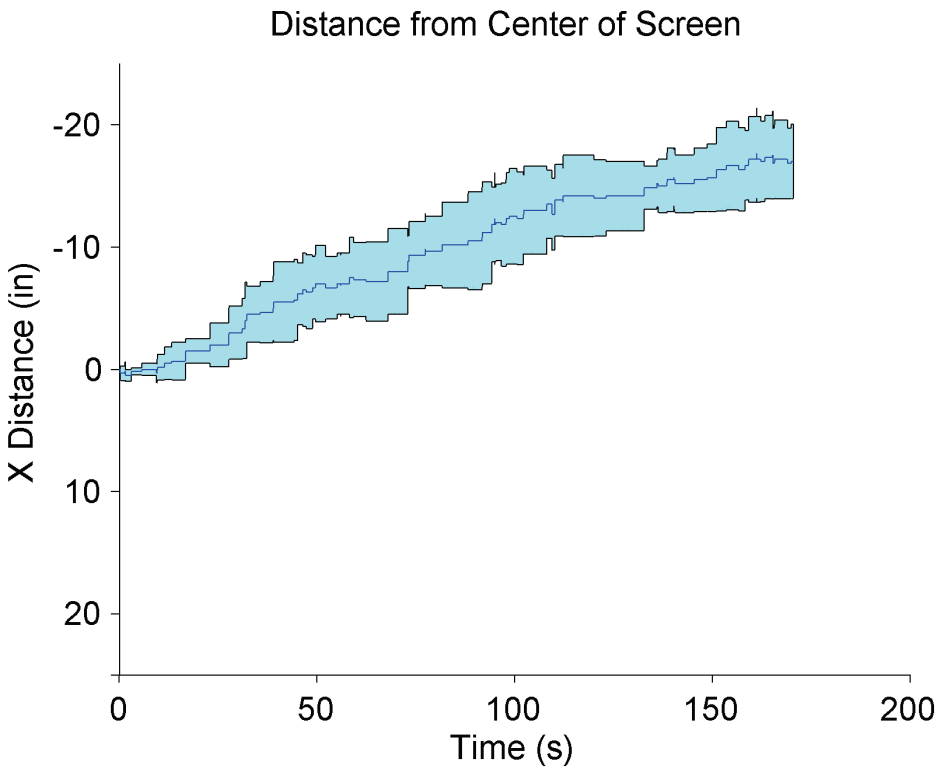


Figure 16: Parallel Parking Effect: Preliminary experiments showing the wheelchair trainer translates left over time

Due to how the trainer is capable of moving and this parallel parking insight, I was able to put together an effective controller. I artificially imposed a depth error to correct for the parallel parking effect based on the X translation. A value, ImposedX, is created to be added or subtracted from the found wall distance value depending on if the trainer is drifting right or left and what direction the trainer is facing. The rest of the code is similar to the code when finding the distance to the wall with a screen present. The only difference is that the correction factor for X translation is subtracted if the trainer is turned right and added if the trainer is turned left.

Code to control X position:

```
double GainX = .5;
double ImposedX = GainX * XTranslation;


for (int pix = xpix; pix >= 50; pix--)
{
   if (AvePix[pix] != 0 && XPos[pix] != 0)
   {
      if (heading >= 0 && heading < 60)
      {
         WallAng = heading;
      }
      if (heading <= 360 && heading > 300)
      {
         WallAng = heading - 360;
      }
      double headingRad = WallAng * (Math.PI / 180);
      AddedXVal = Math.Sin(headingRad) * XPos[pix];
         if (heading >= 5 && heading < 60)
         {
            if (AddedXVal > 12 || AddedXVal < 0)
            {
               AddedXVal = 0;
            }
            WallDistNG = (AvePix[pix-2] * Math.Cos(headingRad)) - AddedXVal - ImposedX;
            break;
         }
         if (heading <= 355 && heading > 300)
         {
            if (AddedXVal < -12 || AddedXVal > 0)
            {
               AddedXVal = 0;
            }
            WallDistNG = (AvePix[pix] * Math.Cos(headingRad)) - AddedXVal + ImposedX;
            break;
         }
   }
}
```

6. Experimental Methods

I did not have an appropriate TV screen to use on a wall. However, I discovered that any glass will work the same way as a TV screen, so I ended up using a window. I opened the blinds so that the width of the window, or 'screen' if you will, visible was 27 inches, a common size for a medium television. During the controlled trials the trainer was positioned at 100 inches from the wall and was also given this distance as the desired distance. For the non-controlled trials the trainer was started at

120 inches from the wall. This was because it would run into the wall within three minutes if started at 100 inches.

From previous research in our lab[13], it was possible to turn off the joystick signals for Y translation. This was done to show that no input other than the trainer on its own or the controller were moving it forward or backward in relation to the wall. Also from previous research[13], it was possible to constrain the trainer so it would only be allowed to rotate a predetermined angle to either side of zero. I set this to 15 degrees as I noticed that the compass would lag, allowing the trainer to sometimes reach angles of 30 degrees from zero. The actual trials lasted three minutes from when I started turning the trainer. Six trials were run with the control algorithm working and six trials were run with the control algorithm off.

RESULTS

1. Experimental Methods Results

Figure 17 shows the Z distance of the trainer (i.e. forward/backward distance from the wall) with and without the control algorithm active. The red line shows the average of the controlled trials and the blue line shows the non-controlled trials average. It can be observed that the trainer stays at around 100 inches, the predetermined set point, from the wall in the controlled case and that the trainer steadily moves toward the wall in the non-controlled case. The standard deviation is also shown on this graph, in light magenta for the controlled group and in light aqua for the non-controlled case. The average standard deviation for the controlled group was 3.14 inches, while the average standard deviation for the non-controlled group was 9.45 inches.

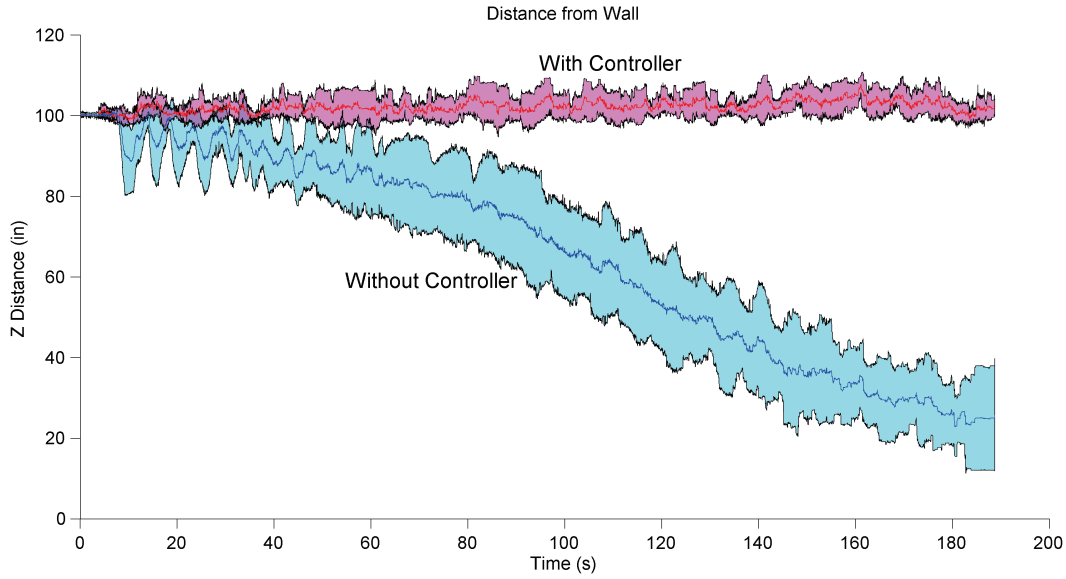


Figure 17: Results showing Z distance of trainer in relation to wall with and without the controller in place

Figure 18 shows the average of X translation (i.e. left/right motion) of the six runs performed. The trainer now stays within an acceptable range and does not exhibit the parallel parking effect. The standard deviation is also shown in aqua. The average standard deviation is 2.35 inches.

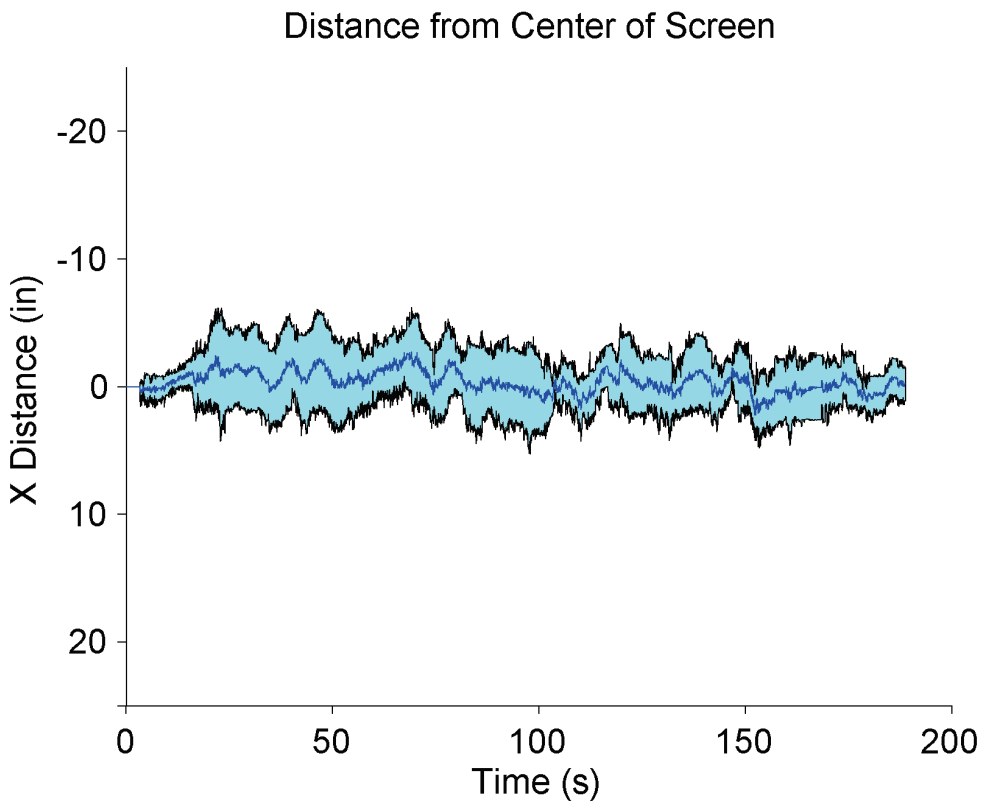


Figure 18: Figure showing X translation average of 6 runs

2. Unexpected Results: Washout Filter

One interesting side-effect of the controller is that it allows the chair to move a small amount forward when the user pushes the joystick forward before returning it to the desired distance from the screen (Figure 19). This behavior is similar to the behavior of a "washout filter" used in motion simulation, such as flight simulators. By allowing a forward acceleration, followed by a slower backward return to the desired forward/backward position, a washout filter conveys a more realistic sense of translation motion.

For the washout filter trials, the desired distance was also set to 100 inches. The translation of the trainer was left on, as this was necessary for performing the experiment. The control algorithm was turned on. I moved the trainer forward as much as the control algorithm would allow and held it for a few seconds, then released the joystick and was brought back to approximately 100 inches. This movement was performed 10 times each trial and four trials were performed.

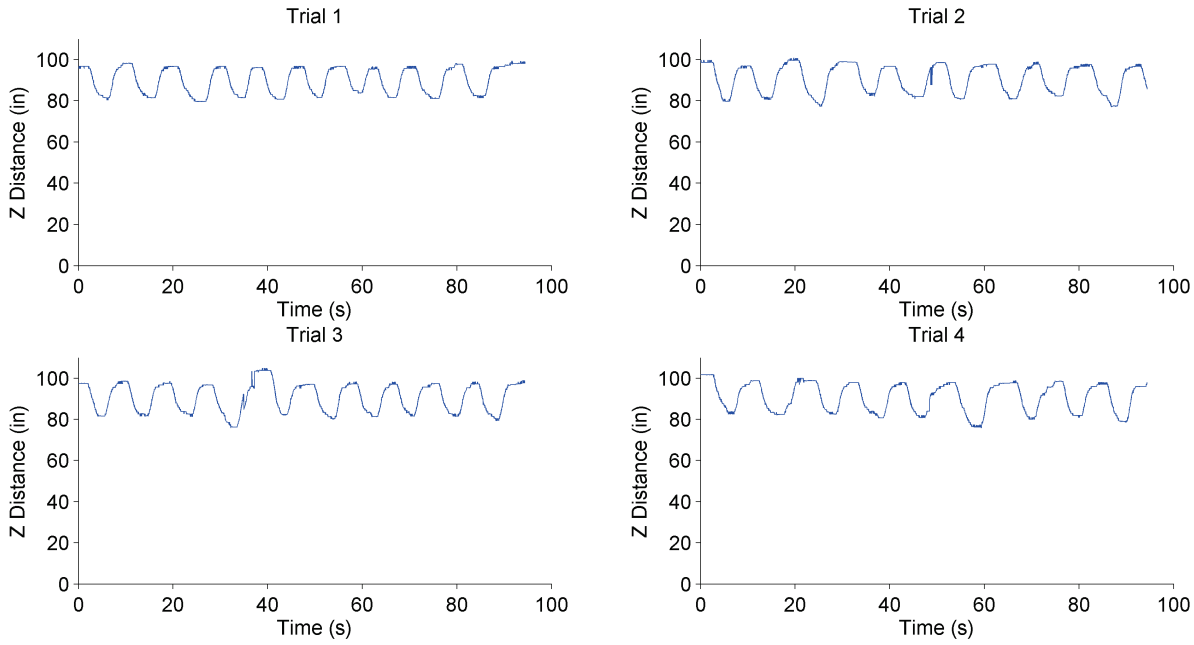


Figure 19: Washout filter effect: four trials were performed

DISCUSSION

The outcome of the final control algorithm is encouraging. The control algorithm kept the trainer at the desired distance from the wall during gaming mode, whereas the trainer would have eventually ran into the wall previously. We can also see from Figure 18 that the trainer can readjust itself to stay in approximately the same location in regards to the X translation over time, and no longer exhibits the parallel parking effect; in fact, it in effect uses parallel parking motions as a control input to the chair, based on left/right position feedback from the Kinect, to actively cancel out any "unintentional" parallel parking. This algorithm is a viable solution as it keeps the KWIC trainer in a desired location during gaming mode.

Both the Z and X distances are sensed robustly. The Z and X distances are constantly being evaluated using the compass angle and the distances the Kinect calculates. The trainer will only act on these values if a therapist instructs it to do so. Even in that situation, the trainer will do several checks first to ensure that the desired wall distance and angle are within an acceptable range to operate. If the therapist entered a desired distance that was unsafe or outside the operating range of the Kinect the control algorithm would not engage. Also if the angle of the compass differs from +/- 5 degrees from zero, the control algorithm will not engage. This is to ensure that the therapist has reset the compass to zero recently so that the trainer will not accidentally find wrong distances. There is also a safeguard within the program so that if the Z distance found is determined to be an unsafe or inoperable value, it is automatically set to the desired distance so as not to move using incorrect information.

The creep found in both Z and X directions were corrected using this algorithm. However, the means by which they were corrected are quite different due to the system being nonholonomic. For the Z direction, the actual distance to the wall was calculated and if this was different than the desired distance to the wall, then a proportional controller would make the trainer move forward or backward

depending on if it were further or closer than desired. The proportional controller changed the amount of torque the KWIC trainer received in the forward/backward direction.

Since the trainer only turns right and left and cannot directly move right and left, a different scheme had to be devised for the X direction controller. I learned from previous experiments that the trainer could be moved right and left using a parallel parking effect. This means that when the trainer is turned in one direction, have it move backward to create a net X direction motion, and when the trainer is turned in the other direction have it move forward, or vice versa. This will effectively move the trainer in either direction along the X axis while keeping it at a fairly stable Z distance. Using this information, a proportional controller was devised that would determine if the trainer was not centered on the screen and add or subtract distances from the found Z distance depending on if it were moving in the positive or negative X direction and what direction the trainer was facing. This effectively works to keep the trainer relatively centered on the screen.

The overall control system is very robust. However, there are elements worthy of further research. The Z and X translation distances found are not always correct and further work could be put into determining the reason for this and make it more accurate. The Kinect currently only finds X_{tr} when the trainer is turned right and X_{tl} when the trainer is turned left (See Figures 12 and 13). This is due to a combination of difficulty in determining calculations and various incorrect values found by the compass and Kinect. This could also be evaluated and a means to find these variables constantly may be found.

There were several unintended consequences of making this controller. The first was discovering that the trainer would move right or left when kept at the intended distance from the wall, or the parallel parking effect. When I understood how this worked, a means of controlling this was able to be implemented. An exciting consequence of this was finding that there were forward and backward accelerations possible with this controller. For example, if you are pressing forward with the joystick there will be a distance where the controller will not allow you to move forward any further due to the

added torque being exactly opposite to the amount you are applying. Then when you release the joystick, there is a movement backward that you experience that could be felt as a deceleration. The means by which drivers experience artificial motion is commonly referred to as washout filter[18]. Many motion based simulators, most notably flight simulators, employ washout filters to make the game experience feel more real to the user. This could potentially be used in the racing game.

CONCLUSION

The implications of this controller are important for both the therapist and the user of the KWIC trainer. The therapist does not need to constantly supervise the child while they are playing videogames in gaming mode because they do not need to worry about the trainer running into the wall any longer. The therapist will need to do a few additional steps in order for the controller to be in place, but the few extra seconds required will be worth it for the advantages gained. What this means for the user is that they will be able to play for longer intervals of time unsupervised, thus allowing them to be more independent. The user will also be confident that the trainer will stay in an appropriate position as they play the chosen game.

It was also important to understand what the X creep was doing in order to control it. In this way the fact that the trainer crept left when the Z control was in place helped evolve the controller. The overall control algorithm would not be as robust as it is currently without this.

The KWIC trainer now stays a certain distance from the wall and resists drastic movement right and left, whereas before it was creeping toward the wall during gaming mode. This is definitely an improvement as far as movement of the trainer. One could argue that this would not enhance the learning of the user in that they may be confused as to how the trainer is moving. For example, the trainer may move right and forward when they turn the joystick right or it may move left and backward when they turn the joystick left. However, it will also give them more autonomy in that they will be able to play games for a longer period of time without constant supervision.

There are many opportunities for future research with the KWIC trainer. Firstly, new hardware would be an enhancement, as this was a large source of error and the next generation would be more maneuverable. Other switches that can be exchanged for the current joystick would also be a huge improvement, as many children in need of a wheelchair cannot use a traditional joystick. Lastly, a

means to use the washout filter found in the controller algorithm would be exciting to implement in some of the games the children play in gaming mode.

REFERENCES

[1] Galloway, James C. Cole, Ji-Chul Ryu, and Sunil K. Agrawal. "Babies driving robots: self-generated mobility in very young infants." *Intelligent Service Robotics* 1.2 (2008): 123-134.

[2] Kermoian, Rosanne, and Joseph J. Campos. "Locomotor experience: A facilitator of spatial cognitive development." *Child development* (1988): 908-917.

[3] Jones, Maria A., Irene R. McEwen, and Laura Hansen. "Use of power mobility for a young child with spinal muscular atrophy." *Physical Therapy* 83.3 (2003): 253-262.

[4] Guerette, Paula, Donita Tefft, and Jan Furumasu. "Pediatric powered wheelchairs: results of a national survey of providers." *Assistive Technology* 17.2 (2005): 144-158.

[5] Balcells, A. Civit, and Julio Abascal González. "TetraNauta: A wheelchair controller for users with very severe mobility restrictions." *3rd Annual TIDE Congress*. 1998.

[6] Connell, Jonathan, and Paul Viola. "Cooperative control of a semi-autonomous mobile robot." *Proc. IEEE Int'l Conf. on Robotics and Automation*. 1990.

[7] Simpson, Richard C., et al. "NavChair: an assistive wheelchair navigation system with automatic adaptation." *Assistive Technology and Artificial Intelligence*. Springer Berlin Heidelberg, 1998. 235-255.

[8] Yanco, Holly A. "Wheelesley: A robotic wheelchair system: Indoor navigation and user interface." *Assistive technology and artificial intelligence*. Springer Berlin Heidelberg, 1998. 256-268.

[9] Nisbet, Paul, et al. "'Smart'wheelchairs for mobility training." *Technology and Disability* 5.1 (1996): 49-62.

[10] McGarry, Sarah, Lois Moir, and Sonya Girdler. "The Smart Wheelchair: is it an appropriate mobility training tool for children with physical disabilities?." *Disability and Rehabilitation: Assistive Technology* 7.5 (2012): 372-380.

[11] Marchal-Crespo, Laura, Jan Furumasu, and David J. Reinkensmeyer. "A robotic wheelchair trainer: design overview and a feasibility study." *Journal of neuroengineering and rehabilitation* 7.40 (2010): 1-12.

[12] Marchal-Crespo, Laura. *Haptic Guidance for Enhancing Human Motor Learning: Application to a Robot-Assisted Powered Wheelchair Trainer.* Diss. University of California, Irvine, 2009. Ann Arbor: ProQuest LLC, 2009.

[13] Secoli, R., D. Zondervan, and D. Reinkensmeyer. "Using a smart wheelchair as a gaming device for floor-projected games: a mixed-reality environment for training powered-wheelchair driving skills." *Studies in health technology and informatics* 173 (2011): 450-456.

[14] Hu, Tiemin, et al. "A neural network controller for a nonholonomic mobile robot with unknown robot parameters." *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*. Vol. 4. IEEE, 2002.

[15] Ding, Dang, et al. "Analysis of driving backward in an electric-powered wheelchair." *Control Systems Technology, IEEE Transactions on* 12.6 (2004): 934-943.

[16] Wang, Hongwu, et al. "An experimental method for measuring the moment of inertia of an electric power wheelchair." *Engineering in Medicine and Biology Society, 2007. EMBS 2007. 29th Annual International Conference of the IEEE*. IEEE, 2007.

[17] "Coordinate Spaces." Microsoft Developer Network. , Microsoft, 2014. http://msdn.microsoft.com/en-us/library/hh973078.aspx. Web. 19 Aug. 2014.

[18] Barbagli, Federico, et al. "Washout filter design for a motorcycle simulator." *Virtual Reality, 2001. Proceedings. IEEE*. IEEE, 2001.