MEASUREMENT OF CONTROL PARAMETERS FOR OMNIDIRECTIONAL TREADMILLS USING RGBD CAMERA

by

Mohammad Azim Ul Ekram

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Engineering,
Concentration: Electrical Engineering
August 2017

Committee Members:

Semih Aslan, Chair

William Stapleton

Bahram Asiabanpour

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

To my parents and sister, who are there for me in every steps of my life. Even if they live far away, they still are the guiding light for me.

AND

To my life partner, Mahmuda Akter Monne, for her support, encouragement and companion which gave me the strength to survive all the stress and frustrations even in the darkest and difficult of days.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Semih Aslan for giving me the opportunity to work in such a promising area. He continually conveyed a spirit of encouragement in regard to research and life. His understanding, wisdom, patience, guidance and encouragement pushed me this far. Without his guidance and persistent help this dissertation would be far from complete.

I also express thanks to my thesis committee members, Dr. William Stapleton and Dr. Bahram Asiabanpour. With their patience and guidance, they supported me all the way till the end and made this thesis possible.

I would also like to acknowledge, Dr. Jesus Jimenez, who allowed me to setup equipments on his lab, Tasnuva Udita for her constant reminders to work diligently, Syadus Sefat for his brilliant review, Texas State Writing center for their awesome grammatical reviews.

Finally, I acknowledge my wife, whose companion blessed my life with joy specially during the endless nights of work.

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

LIST OF ABBREVIATIONS

Abbreviation	Description
COM	Center-of-Mass.
GPU	Graphics Processing Unit.
HMD	Head Mounter Display.
MSE	Mean-Squared-Error.
OBDP	Omni-directional Ball-bearing Disc Platform.
RGBD	Red-Green-Blue-Depth.
SDK	Software Development Kit.
ToF	Time-of-Flight.
VE	Virtual Environment.
VR	Virtual Reality.

ABSTRACT

Omnidirectional treadmill systems are an effective platform that allows unconstrained locomotion possibilities to a user for effective VR exploration. There are two most common problems associated with Omnidirectional treadmill systems. First one is the mechanical design of it. The second one is controlling algorithm that controls the treadmill. This thesis focuses on the second problem and presents an algorithm which measures important parameters for controlling the direction and speed of omnidirectional treadmills. The primary objective is to collect skeleton data from Kinect Sensor(RGBD) and measure speed, acceleration and orientation vectors of lower body joints. From these measurements, it is possible to calculate the control parameters for omnidirectional treadmills. Using these parameters, the treadmill will try and compensate user motion, to keep the user close to the platform center. Another objective is to validate the parameters found from the algorithm and determine the accuracy of the algorithm using Kinect camera. Also, this article explores whether the kinect can be a viable replacement for current motion capture systems used for this purpose. Usage of Kinect camera can make VR experience non-invasive and low-cost.

I. INTRODUCTION

Motivation

Virtual Reality (VR) has received much attention and significant development in the past decade. In terms of graphics quality, 3D displays and haptic feedback interfaces [1, 2, 3], VR is now considered to be the next platform for gaming, skill training, education and medical research. High-fidelity driving [4], flight simulators, combat simulators as well as various multipurpose simulators [5] have been developed to navigate through the Virtual Environments (VEs). These research and developments shaped VR as an essential tool in many diverse areas such as operation theater [6], military skill training [4], rehabilitation [7] and various other medical analyses [8, 9]. In addition, VR is also spreading into domains such as gaming, education and architecture. Although VR headsets have been around for a decade, there are few pure VR locomotion interfaces that have been developed which contains active locomotion through large VEs.

Our daily life depends intuitively on our ability to navigate through the world on foot. As a matter of fact, throughout the largest part of evolutionary history, the only mode of transportation of human ancestry was walking. Current VR setups, however, lack the essential feature of walking through VEs or simulate only in a very restrictive manner. Keyboard, mouse, joystick, or similar input devices are used, in most cases, to simply navigate through the VE. This creates a sensory conflict, where the user is physically not moving, but receives visual input congruous with self-motion. According to various behavioral studies, this sensory conflict may impede the formation of an accurate spatial-temporal representation of the environment and impede navigation performance [10] and even cause simulator sickness [11]. Hence, an omnidirectional locomotion interface is required in order to allow users to walk naturally through large-scale

VEs. In principle, a general solution is offered by treadmills, where the users can walk through arbitrarily large VEs while keeping them in a relatively restricted area. However, general treadmill solutions don't let you walk in any direction. Treadmills are unidirectional.

The treadmill solution is far from satisfactory because of two major problems. Firstly, the treadmill solution allows users to walk forward direction only, restricting the possibilities of 360° rotation. This is the mechanical design problem which is an active research field. There are several design patents filed with different mechanisms [12, 13].

The second major problem is controlling the velocity of treadmills as a function of the user's walking characteristics [14]. Users should be kept on the treadmill while either walking or standing still for obvious reason. Therefore, most available setups permit the user to walk at only one fixed speed, predefined by the treadmill. A desirable solution would be for the treadmills to respond to changes in walking speed and direction, but this poses some serious set of problems. Too high acceleration of the treadmill can disrupt the immersiveness of VR and can even bring the user out of balance. On the other hand, too low acceleration would make the person walk off the treadmill.

There are several solutions to the second problem, which are cited in Chapter II. Both the control and design of omnidirectional treadmills have been done with precision equipment and expensive systems. This paper examines the possibility of utilizing Kinect camera for the purpose of controlling omnidirectional treadmills.

Research Questions and Goals

This article focuses on the second problem of controlling the speed of omnidirectional treadmills in such a way that doesn't disrupt the VR immersive experience. For employing an unconstrained omnidirectional treadmill, achieving natu-

ral walking motion is imperative. This puts several constraints and requirements on the treadmill system in terms of its size, possible speeds and accelerations. These are all important parameters which play key roles in controlling the omnidirectional treadmill [14].

Our knowledge of omnidirectional treadmill is largely based on very limited data as research in this area is yet to be matured. The aim of this research is thus to develop a system for the omnidirectional treadmill to calculate controlling parameters (velocity, acceleration, orientation, reference points) from Red-Green-Blue-Depth (RGBD) cameras and allowing for changes in both walking speed and direction. In this article, we describe an algorithm and report the results of experiments that we conducted to evaluate its effectiveness.

Hypothesis

Instead of using expensive motion capture systems it is possible to use Microsoft Kinect RGBD cameras with reasonable accuracy and speed to control the omnidirectional treadmill for users to experience natural walking motion. Controlling the omnidirectional treadmill requires calculation of speed, acceleration, orientation and reference points based on Kinect depth and body joint data.

Questions and Goals

The overarching goal of this research is to determine whether low-cost RGBD cameras, like Microsoft Kinect, can be used effectively to control an omnidirectional treadmill. A number of sub-topics are explored in order to answer this question.

- What SDKs will be used with Kinect hardware as several is available? What kind of data and information are available from these SDKs which can be used for this research purpose?
- What is the quality and precision of the joint data acquired from Kinect?

What is the sampling rate and consistency?

- Calculate velocity, acceleration and reference point based on joint data.
- How the control algorithm works with the consistency of Kinect?
- Validation and precision measurement of the control algorithm.

In order to answer these questions and meet the goal, a prototype application was developed and used to gather data and visualize the body joints and show important information under various conditions.

Outline of Thesis

This thesis consists of six chapters. Introduction, Related Work, Technical Details, Methodology, Experiments - Results and Conclusion.

The thesis begins with the first chapter, giving an introduction to the overview of thesis, it's importance and significance of thesis contributions. In second chapter, related literature review is presented. Although exact match of this work is very few, there are other similar research which are mentioned in this chapter.

Third chapter is the longest chapter. It contains a lot of technical details required for the methodology chapter. Details about Kinect camera, it's specification, distance measurement, basic gait parameters are described in this chapter.

Methodology is illustrated in chapter four. This chapter introduces the proposed algorithm and system setup. Several fundamental topics are represented such as joint smoothing, floor parameter calculation, speed and acceleration calculation etc.

Chapter five is the experimental result chapter, where the experiments are described and subsequent data-sets are depicted and described. Also, it contains opinions and explanation of various measurements.

The final chapter entails the conclusions of the thesis and includes limitation of the thesis and provides useful insight into suggestions for improvements and future plans.

II. RELATED WORKS

Since the advent into VR in 1965, special Head Mounter Display (HMD)s are the most common way to experience VR. In 2003, Fernandes et al. developed *Cybersphere*, which is an immersive spherical projection system. It comprised of a large, hollow and translucent sphere ideally 3.5 meters in diameter and is supported by means of a low-pressure cushion of air. This is depicted in Fig. II.1. Walking movements of the observer cause the sphere to rotate (see Fig. II.1), al-

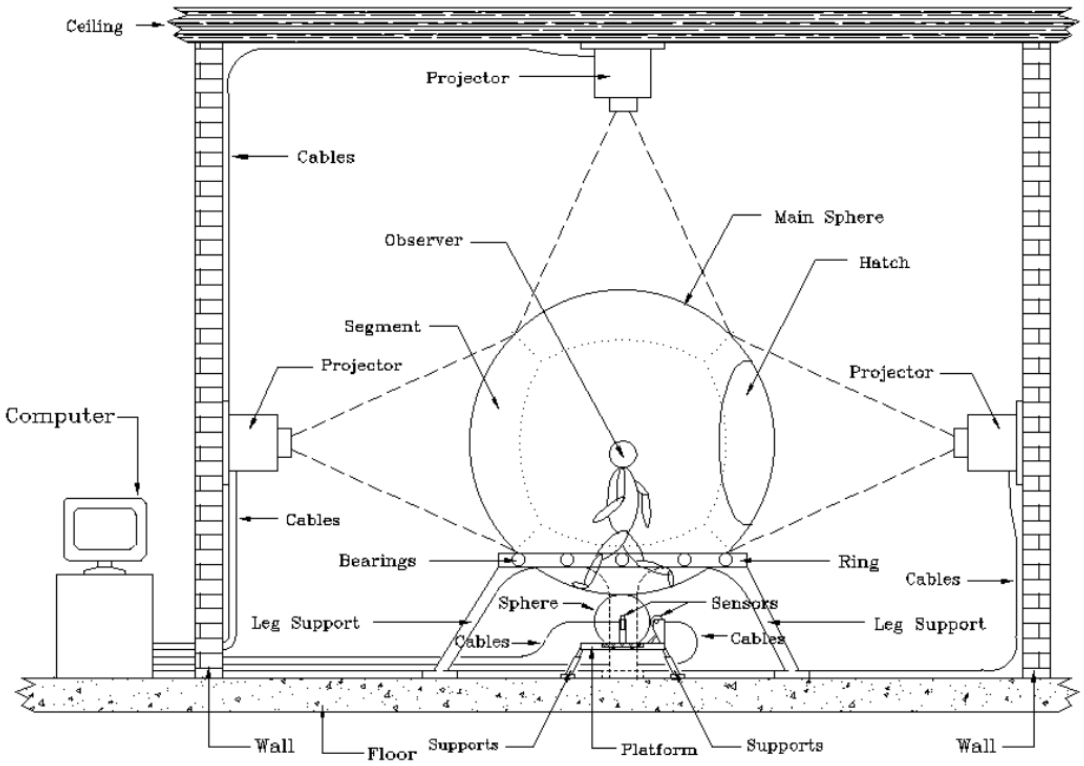


Figure II.1: Proposed Schematic of the Cybersphere [15]

lowing the user to navigate and explore the virtual world in a natural manner according to his or her interest in the visualized domain. Movement of the *Cybersphere* is dictated by the walking motion of the user in all directions [15]. Images are projected onto segments of the outer surface of the large sphere by high power projectors. Apart of the complex structure and design, it was difficult to project the images in a multi-viewpoint fashion and natural movements are also

very limited, as a natural locomotion through the virtual environment is usually restricted to the currently shown spot of the scenario. In the same year, a stoll-based omnidirectional VR interface was brought to attention of research community. It implements an unique locomotion design. Rather than using a treadmill, the author used a locomotion mechanism called Omni-directional Ball-bearing Disc Platform (OBDP), which allows the user to walk naturally on it and thus to navigate the virtual environment. The gait sensing algorithm that simulates the user's posture based upon his footstep-data collected from the OBDP is then elaborated, followed with an omnidirectional stroll-based virtual reality system to integrate the OBDP with the gait sensing algorithm. Significantly, instead of using the three-dimensional (3-D) tracker, the OBDP adopts arrays of ball-bearing sensors on a disc to detect the pace. No other sensor, except the head tracker to detect the user's head rotation, is required on the user's body [16]. Based on that design, similar products in the market where instead of using ball-bearings, strolls under the shoe were used by *KatWalk VR* [17], *Virtuix Omni* [18], *Cyberith Virtualize* etc. The design, while allowed the user to stay in the middle of the platform and run or walk, didn't employ a natural experience. Also, the user stance was limited and practice was required for proper use. MSEAB Weibull Company created the *Virtual Theater* for military purpose, which was later used for academic and research purpose. User can move around in a fixed speed, but the user rotating direction is not handled very well [19]. Recently, *Infinadeck* showcased an omnidirectional treadmill prototype in CES 2016. While that promotes very good VR experience with natural walking motion, it requires a belt to be wrapped around the torso which can sense the direction and force of body movement and thus can suffer from impeded navigation performance and simulator sickness described above.

To the best of knowledge, *CyberWalk*(filed for patent in Germany [14, 20]) was the only unconstrained non-invasive omnidirectional treadmill we found, which didn't require any special devices to attach to the torso [14]. The ba-

sic concept diagram of the *CyberWalk* platform is shown in Fig. II.2. Three

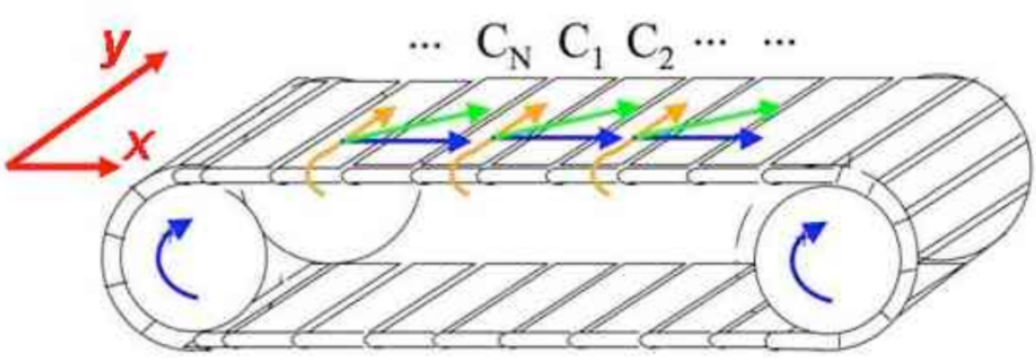


Figure II.2: Kinematic Concept of 2D CyberWalk platform [13]

head pointers are attached to the headset. *CyberWalk* platform detects the head pointers using *Vicon Motion Capture System*, a very costly motion detection system. Then, based on these pointer positions and second-degree control equations, they calculate possible speed and acceleration to control the omnidirectional treadmill using a separate PC. *CyberWalk* can be categorized as a complete unconstrained omnidirectional treadmill, but it uses very costly equipment and software to provide this natural experience. Their primary omnidirectional treadmill design weigh more than 10000Kg and has a moving mass of approximately 7500Kg [13]. There is a smaller version of it described in [20]. Basic design of the omnidirectional treadmill system is shown in Fig. II.3 and the overview of control algorithm for *CyberWalk* is shown in Fig. II.4.

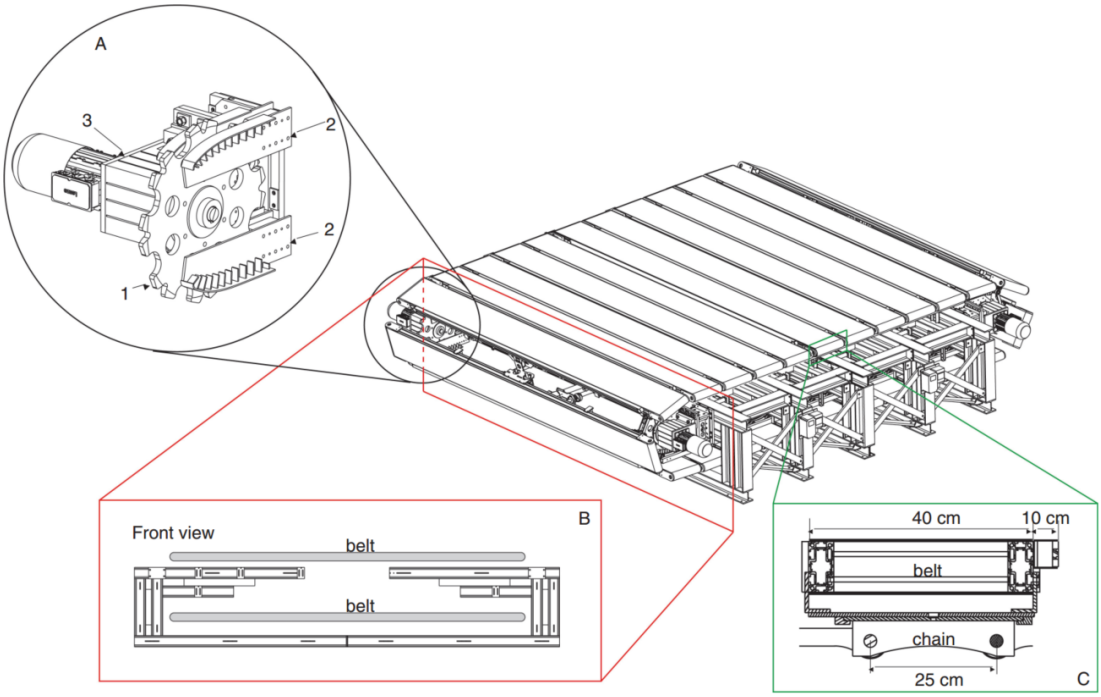


Figure II.3: CyberWalk Treadmill Design described in Souman et. al. [13]

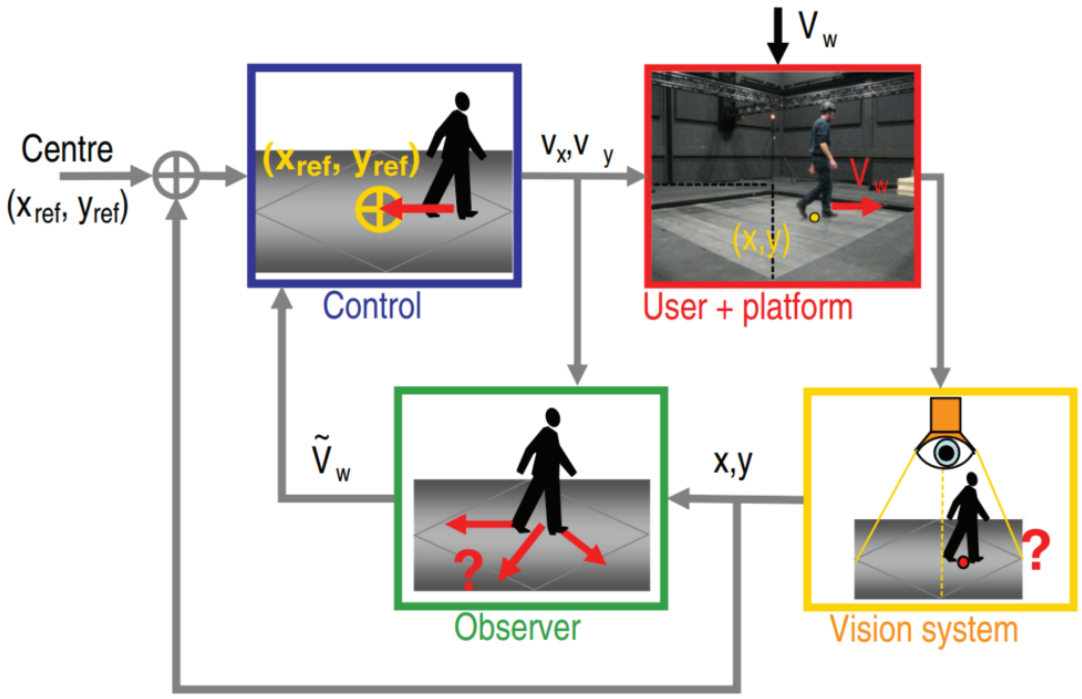


Figure II.4: CyberWalk treadmill higher level control system [13]

III. TECHNICAL DETAILS

Kinect

General Overview

Popularity of RGBD cameras have been significantly raised within research communities, mainly due to the emergence of Microsoft Kinect series. Kinect is a RGBD acquisition device designed by Microsoft as a contact free controller for Xbox. Initially, it was meant only as a major accessory for their Xbox game consoles. In addition to color camera, it also has depth camera which allows to find out both color and spatial information about captured scenes. In 2010, Kinect v1 was released which used the *Structured Light Technology* to capture depth information [21]. The second version of Kinect was released in 2014 along with the new Xbox version. The new Kinect uses a *Modulated Light* or Time-of-Flight (ToF) technology for depth measurements [22]. It enables Kinect to obtain the depth information with much better resolution and quality while also limiting the interference from outside sources [22, 23].

Compared to most 3D scanners, Kinect is very affordable which makes it an attractive tool for many researchers and companies. However, Kinect was originally designed for tracking human body movements. Since the tracking has to be done in real time, Kinect has been designed such that it could capture the depth frames with a reasonable frame rate of 30 Hz maximum. However, this frame rate means that there is very little time for accurate measuring to be done. As a result, the Kinect depth information can be considerably noisy or even incomplete [24]. Nevertheless, researchers have obtained decent results using Kinect as a 3D modeling tool and human skeleton tracker. It features 25 point skeleton map for up to six people. With a wider horizontal and vertical field depth, it has a tracking range from 0.5 meter to 4.5 meters [25, 26, 27, 28].

Depth Camera

The latest Kinect sensor(v2) has three infrared light sources each generating a modulated wave with different amplitudes. In order to capture reflected waves, Kinect also has an infrared camera. Location of lasers and sensors are shown in Fig. III.1.

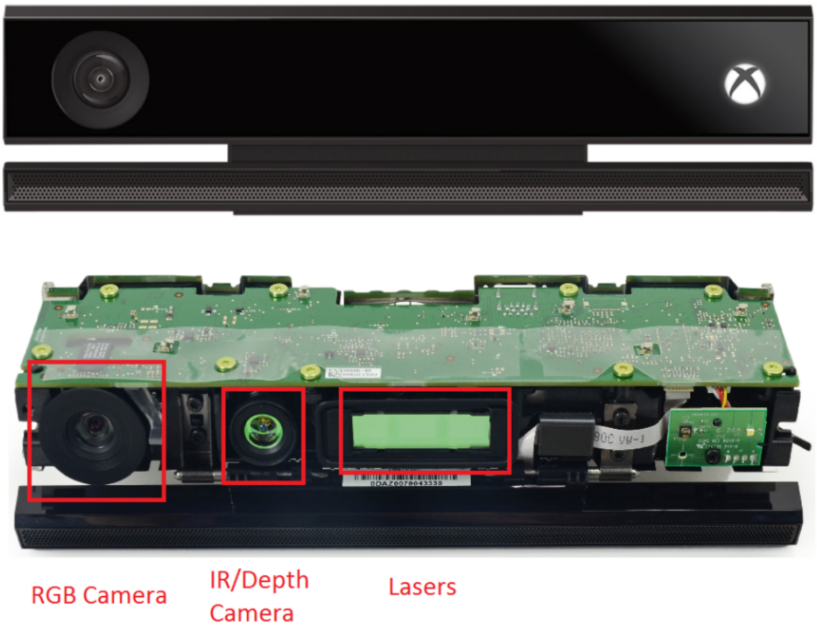


Figure III.1: Stripped down Kinect showing location of sensors [29]

Kinect Specifications

State of the art CMOS array of differential pixels is used for the infrared sensors in Kinect v2. Performance parameters of Kinect Sensors are listed in Table III.1. According to [30], each pixel has two photo diodes (A, B), which are being controlled by the same clock signal that controls wave modulation. Photo diodes convert captured light into current which can be measured. The diodes are driven by the clock signal such that if $A = [a_i]$ is turned on, $B = [b_i]$ is turned off and vice versa. The following properties are observed:

- $([a_i - [b_i]])$ shows correlation between retrieved light and the clock signal and can be used to obtain phase information ("depth image")

Table III.1: Detailed Kinect depth sensor performance parameters

Process Technology	TSMC 0.13 1P5M
Pixel Pitch	10u*10u
Pixel Array	512*424
Chip size	.2mm*14.2mm
System Dynamic Range	>2500 = 68db
Modulation Contrast	68% @ 860nm @50Mhz
Modulation Frequency	10-130Mhz
Average Modulation Frequency	80MHz
FOV	70 (H) X 60 (V) degrees
Depth Uncertainty	<0.5% of range
Distance Range	0.8-4.5m
Operating Wavelength	860nm
Frame Rate	Average 30fps, max 60fps
ADC	2GS/s
Effective Fill Factor	60%
Reflectivity	15%-95%
Chip Power	2.1W
Responsiveness@860nm	0.144 A/W
Readout Noise	320 uV differential
ADC Resolution	10

- $([a_i + [b_i]])$ gives regular grayscale image illuminated by normal ambient lighting ("ambient image")
- $\sqrt{\sum_i([a_i] - [b_i])^2}$ gives grayscale image that is independent of ambient lighting("active image")

As the infrared emits 860 nm wavelength light, a narrowband-pass filter is used to block all light except the 860 nm wavelength range that corresponds to infrared illumination system wavelength. Kinect also uses multi-shutter proprietary engine that accumulates data from multiple shutters and chooses the best shutter value for each pixel. The longest shutter time is chosen which does not cause saturation. The engine also normalizes all values relative to the longest shutter time.

Distance Measurement

As mentioned before, Kinect v2 uses optical ToF technology for measuring distances using a 0.13 μm CMOS system-on-chip for a 512×424 ToF image sensor with multi-frequency photo-demodulation [31, 30]. The operation principle in a ToF device is based on measuring the time it takes for light waves to travel from emitter to object and back to sensor. Because of the reflection, there is a change in phase shift and amplitude. Amplitude is not necessary for calculating distance, hence it is discarded. Kinect sensors use three different phase-shifts of $0°$, $120°$ and $240°$ [23]. Since measuring the distance is based on phase shift of the modulated wave, the maximum uniquely measurable distance depends on the wavelength of this modulated wave. Phase wraps around at $360°(2\pi)$. Hence, using longer wavelengths generally allow for measuring longer distances. Better resolution can be achieved using shorter wavelengths [31]. To achieve both good resolution and measuring longer distances, three different frequencies of $120MHz$, $80MHz$ and $16MHz$ are used by Kinect [32] as shown in Fig. III.2. Common wrap around for these frequencies occurs at 18.75 meters, which is the also the maximum distance for a Kinect device where depth can be identified with a minimum precision [23, 31]. Kinect v2 sensor connectivity drives actually outputs the phase shift values which gives the researchers the opportunity to provide depth map estimation of their own. In the current thesis, the default implementation from Microsoft SDK that produces depth in millimeters for each pixel is used.

Distance to 3D Coordinates

In order to model human skeleton and other objects, true 3D coordinates for the points are desired. This can be achieved using the standard pinhole camera model [33], doing the calculation in reverse order. That is we have the depth or distance or Z-distance of each point.

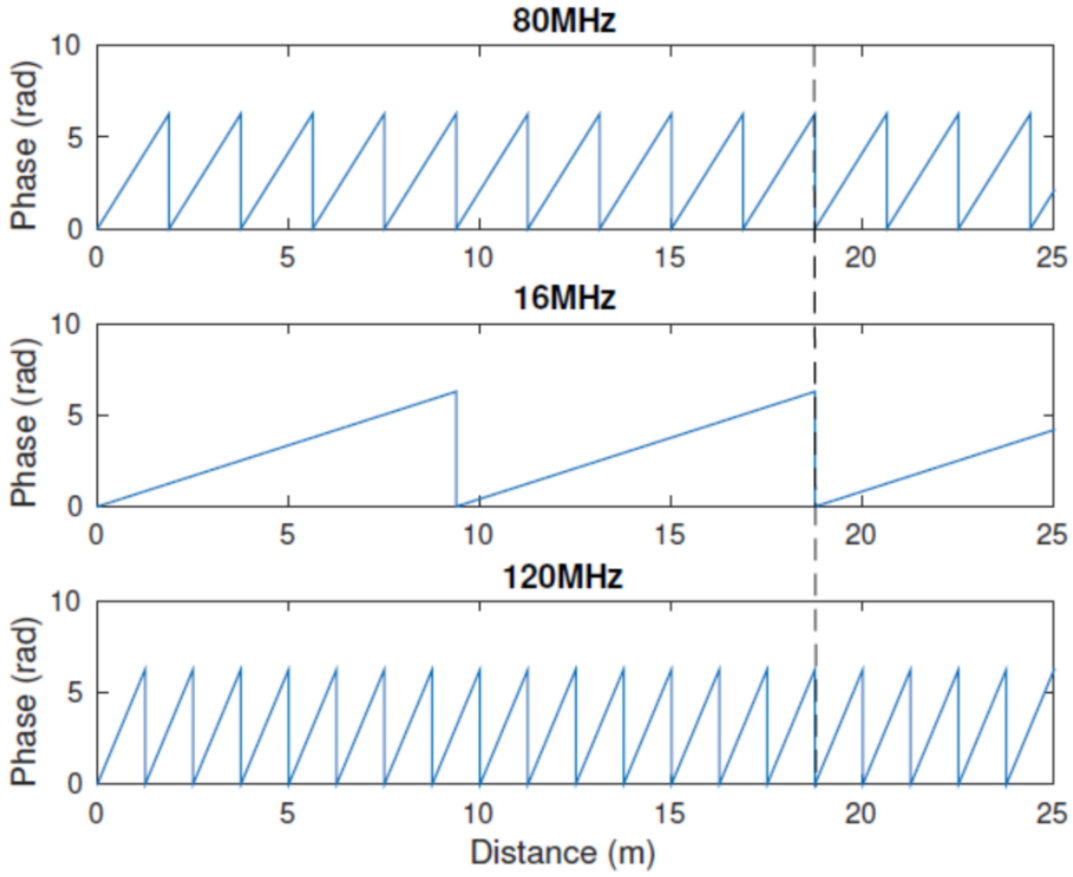


Figure III.2: Wrapped phases and distances for 80MHz, 16MHz and 120MHz modulated waves used in Kinect distance calculation [32]

Given a point P, to calculate distance z from camera center C we'll need distance from principal point to point P, projected on image plane x, camera center distance d from P and focal length f. First, projection of the distance from C to P on image plane can be calculated below in (III.1):

$$l = \sqrt{f^2 + x^2} \tag{III.1}$$

Then, based on the similar triangle property from Fig. III.3,

$$\frac{z}{d} = \frac{f}{l} \tag{III.2}$$

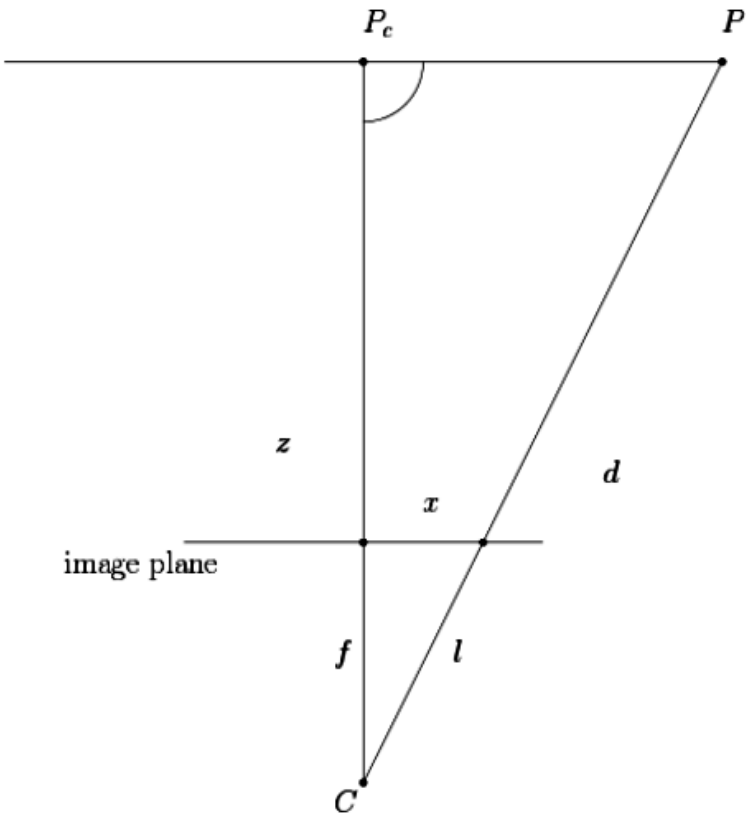


Figure III.3: Real depth Calculation from Distance information

From those above two equations (III.1), (III.2)

$$z = d\frac{f}{l} = d\frac{f}{\sqrt{f^2 + x^2}} \tag{III.3}$$

Kinect SDK [29] outputs the depth values which are already in the depth map format. It means that instead of the distance from camera center to point, each pixel depth value represents the distance to the plane that has the corresponding point and is perpendicular to the camera principal axis [33]. The coordinates (X,Y) can be found from the standard pinhole camera model. Assuming the center of the coordinate system is at camera center, from Fig. III.4, the distance from center of projection to the image plane is f. A line starting from center of projection and perpendicular to the image plane is called principal axis. If $P(X, Y, Z)$ is a 3D world coordinate of a model and $p(x, y)$ is the corresponding points in the image, from Fig. III.3, by similar triangles property, image coordi-

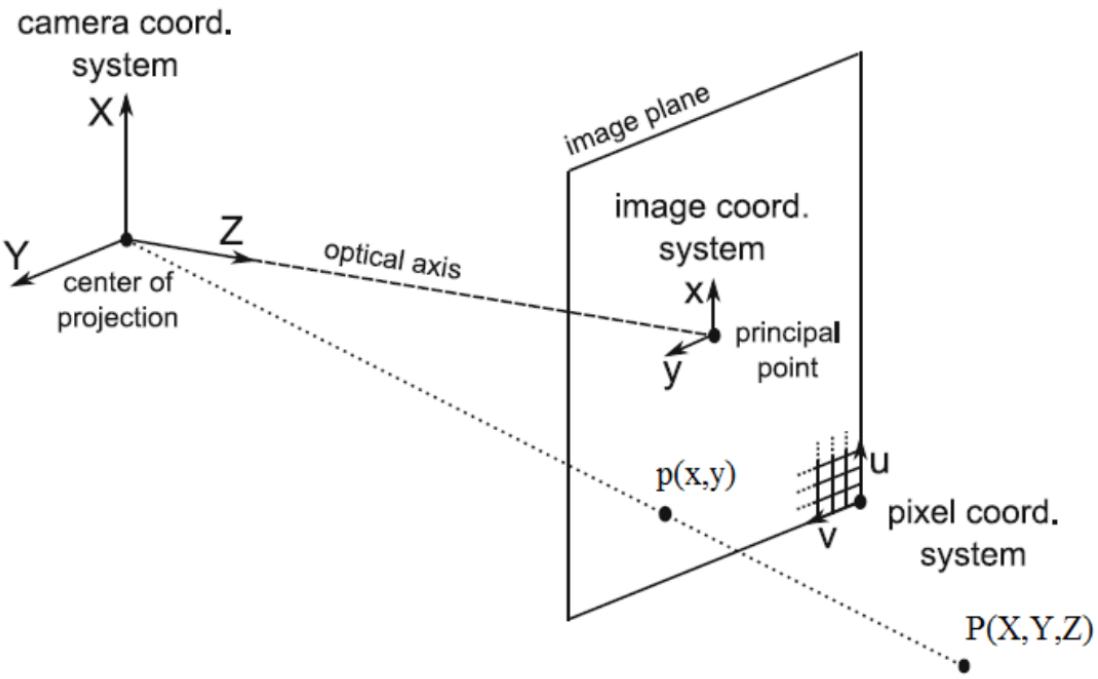


Figure III.4: Pinhole camera model

nates can be written as

$$x = f\frac{X}{Z}, and, y = f\frac{Y}{Z} \tag{III.4}$$

Here, x and y are described in real-world coordinates but in pixel unit. To normalize, column-wise and row-wise density (pixel-per-millimeter) is necessary. After normalizing and translating to the origin location, assuming (x_0, y_0), we have (III.5):

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{X}{Z} \\ \frac{y}{Z} \\ 1 \end{pmatrix} = KP' \tag{III.5}$$

Here, K is the intrinsic matrix, which is fixed for Kinect, (α_u, α_v) represent focal lengths and (u_0, v_0) coordinates of the principal point. Since Z is known, X and

Y can be calculated for all u and v using following relation

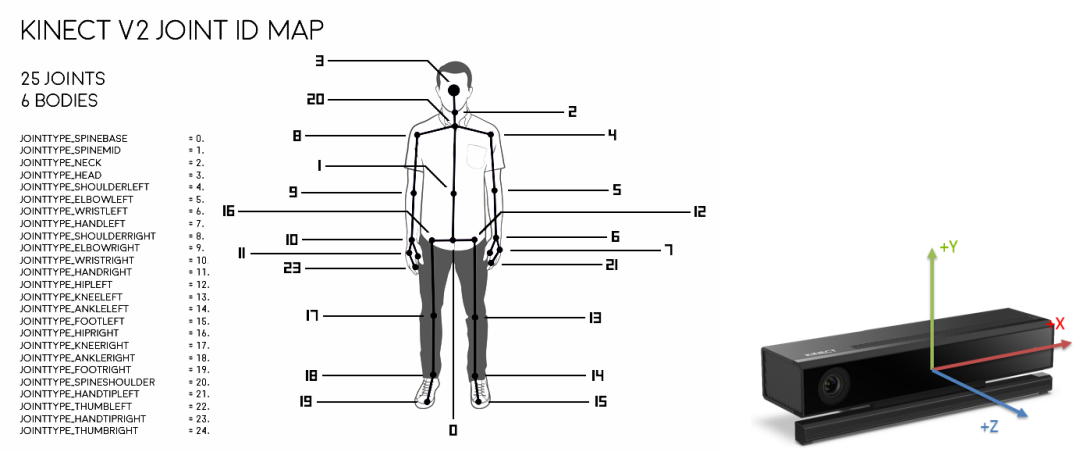
$$ZK^{-1}\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{III.6}$$

Kinect SDK already has an easy to use function which converts between camera-coordinate space and image coordinate space. For OpenNI and other libraries, the function have to be user-defined as these equations depend on external calibration.

Reason for Choosing Kinect

Hardware Precision and Features

It is very important to choose a good RGBD camera which is available, has a good performance to usability ratio, and is vetted by the research community as a viable and low-cost replacement to costly motion capture systems. Microsoft Kinect was chosen as there were numerous research conducted with it, e. g., medical field, military simulator, VR, multimedia avateering, etc. Also, there were several publications indicating its performance and accuracy. Xu et al., Galna et al. executed several evaluative tests of Kinect vs state-of-the art *Vicon Motion Tracking System*. In their evaluation, it was found that the Kinect has good temporal accuracy, hence, step length and stride length were measured with good precision [34, 28]. For this reason, this property will be used in our system. But, unlike temporal accuracy, spatial measurements were not very accurate [34]. As temporal accuracy is very important in determining speed, the localized spatial data can be filtered or predicted using various methods [26, 24]. Detailed reports can be found at the respective articles. Although, the spatial accuracy is not as good as the *Vicon Systems*, many authors have successfully utilized Kinect for various research [35, 36, 37, 34].



(a) (b)

Figure III.5: Skeleton Map and Coordinate system defined by Microsoft Kinect [29]

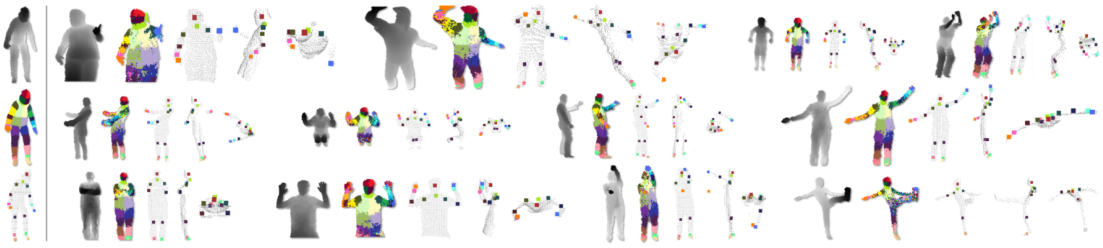


Figure III.6: Segmentation and Learning process used in [38]

Microsoft Kinect v2 features tracking of 25 joints for at most six people. Kinect exploits both temporal and kinematic constraints to approximate joint positions with occlusion handling. Along with temporal data and kinematic constraints, Kinect also partly implements the work of Shotton et al. for real-time human pose recognition from depth images. In article [38], from depth images, body parts are segmented and recognized from depth images as an intermediate representation for human pose estimation, and demonstrated that the classifier can be made invariant to human body shape and pose by training a large corpus of synthetic data [39]. Finally, using an unpublished, proprietary algorithm, a skeleton model is fitted to the hypothesized joint positions. As shown in Fig. III.6, depth images are first segmented then categorized using Randomized Decision Forests [38, 39], which is a well known algorithm for supervised machine learning. Based on the learned data the joints are then approximated.

Multiple Choices of Kinect Dirvers and SDK

The first Kinect release was meant only for Xbox platform. But as it's body tracking and depth sensing capabilities, it soon gained traction in the research community. So several open source and closed source drivers were developed to use Kinect with PC. Later Microsoft also released their own SDK for use with Windows PC [29]. The exploration of other drives other than Microsoft Kinect

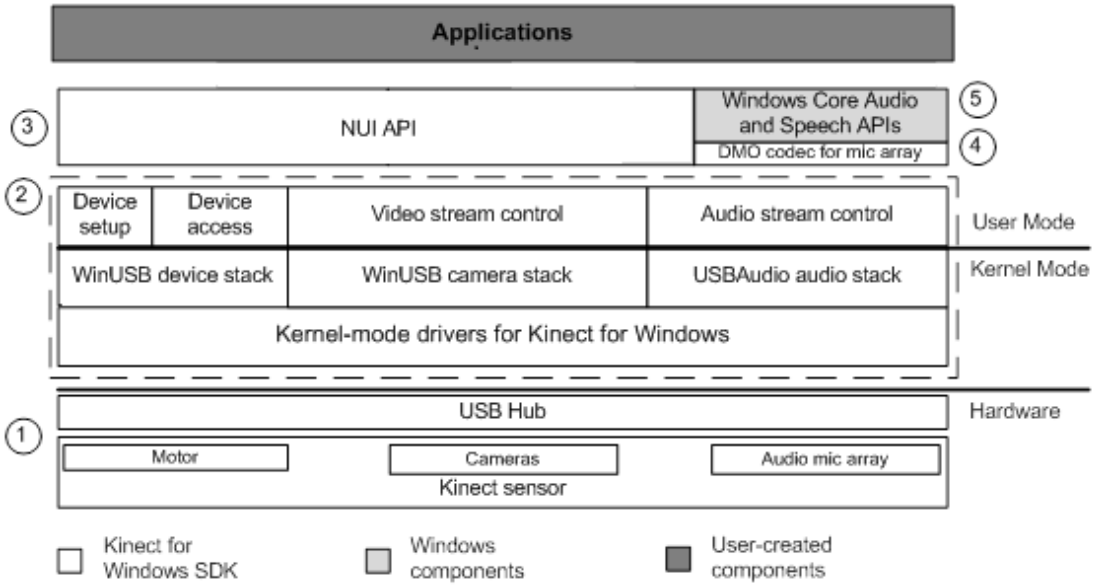


Figure III.7: Overview of Kinect SDK Architecture [29].

SDK is important as they provide open source and platform agnostic utility. Also some of them are faster than Kinect SDK in some areas as they calculate depth and 3D position using different algorithms. Now a day's libfreekinect and OpenNI are most popular within research community who wants raw data from Kinect or requires Linux platform.

Table III.2 below compares most popular open source drivers as of 2017 except Microsoft which is closed source but better performing. For our research need, Microsoft SDK was chosen as it is shown in Table III.2 that Microsoft driver has been pre-calibrated with proper extrinsic and intrinsic parameters. This is a huge advantage as poor calibration can cause jittery or displaced skeletal data, which will hamper calculations. Also, C# language support was

Table III.2: Available Kinect Drivers and SDKs

Library Name	**Platform**	**Features**	**Languages**
libfreenect2	Windows, Linux, MacOSX	- Color and depth images - motor and LED control - Fakenect Kinect simulator - Record to file - GPU Processing	C,C++,Python Java, Lisp, Javascript
ROS freekinect	Unix	- Color and depth images - Motor and LED control	Python, C++
OpenNI	Windows, Linux, MacOSX	- Color and depth images - User identification - Feature detection - Gesture recognition - Joint tracking (calibration required) - Record to file	C, C++
Microsoft Kinect SDK	Windows	- Color and depth images - User identification - Feature detection - Gesture recognition - Joint tracking (no calibration required) - Floor detection - Audio Processing - Easy Application integration - Record to file	C, C++, C#, VisualBasic, F#, Python

a plus point as chosing C# made integration, GUI visualization easier with significantly quicker and less arduous. An argument can be made for performance comparison of C++ vs C# with Kinect camera, but that's rather a discussion beyond the scope of this article. Fig. III.7 contains API and driver hierarchy for Kinect SDK by Microsoft.

Basic Gait Parameters

Human gait is an important indicator of health with application including but not limited to diagnosis and monitoring. Among several proposed and well known methods for gait analysis, marker-based systems which typically uses IR cameras and markers placed on a subject's body are favored for their accuracy. But these devices are very expensive and impractical to move once setup. Kinect has been used extensively since it's debut in 2010 for gait analysis [28, 34, 40] and other medical analysis as mentioned before. Gait parameters typically contain information on different gait cycle parameters, pelvic movements, feet movements, etc. We are only interested in the pelvic and feet movements analysis to determine which joints to focus on in our algorithm and importance of those joints.

The gait cycle is the time between successive feet contacts of the same limbs. Therefore, in Fig. III.8 when the reference(grey) feet contacts the ground and ends with subsequent floor contact of the same feet is called one gait cycle [41]. Gait cycle is divided into two major phases: stance and swing. The stance phase is referred to the period in which the feet is in contact with the ground, and comprises of the first 62% of entire cycle. On the other hand, the swing phase comprises the remaining 38% of the cycle and corresponds to the phase in which the feet does not touch the ground. Being in this moment suspended on the air, the body moves forward in order to induce limb advancement in the locomotion [40]. *Step length* is the distance between the heel contact point of one feet and that of the other feet. *Stride length* is the distance between successive

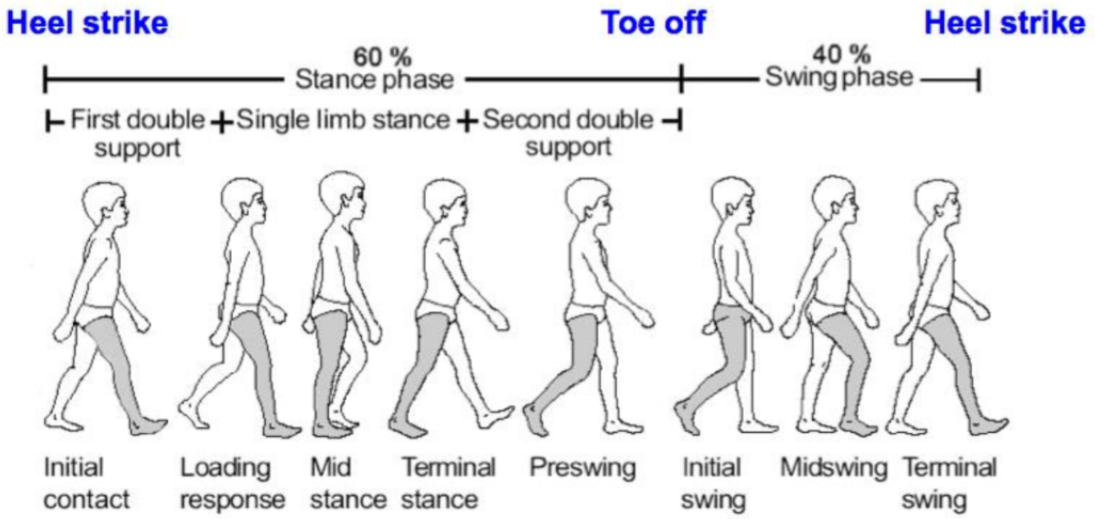


Figure III.8: The complete gait cycle and associated events [41].

heel contact points of the same feet. Hence,

$$StrideLength = 2 \times StepLength \tag{III.7}$$

The *Cadence* defined as the rate of walking expressed in steps per minute, is an important parameter. *Cadence* for a normal person is $100 \sim 115$ steps/minute. Thus in the experiment, we used characters of different stride length and each of them took around 10 steps within six seconds (for $160 \sim 180$ frames in 30fps) for experiment results.

As our algorithm uses three joints, namely two feet and the spine-base, significance of choosing these joints must be stated. As shown in Fig. III.9 it is seen that the Center-of-Mass (COM) trajectory is smooth and have a very little magnitude along Y-Axis. All the magnitude is along Z Axis (forward). Also, notice from Fig. III.10 that, COM also moves very little in X-Axis. As a results, the COM is a very reliable point to measure distance and speed. As a matter of fact, Kinect outputs a joint similar to COM, called "Spine-Base". Later in chapter IV, the "Spine-base" is used for speed and acceleration calculation. Also, notice that in Fig. III.11 there is a region where the body is in stance phase and doubly supported by both legs. It is a phase when one leg has just touched the

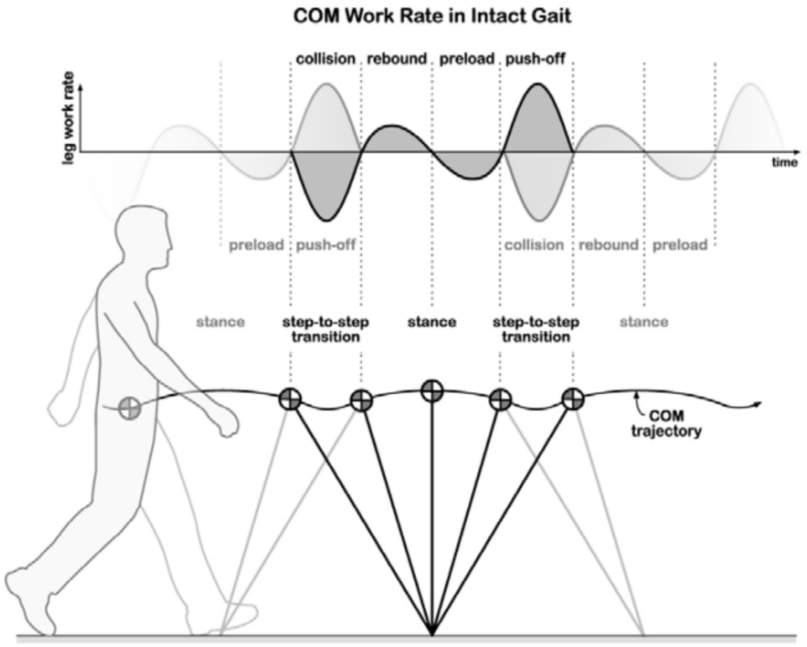


Figure III.9: COM trajectory and step-to-step transition.

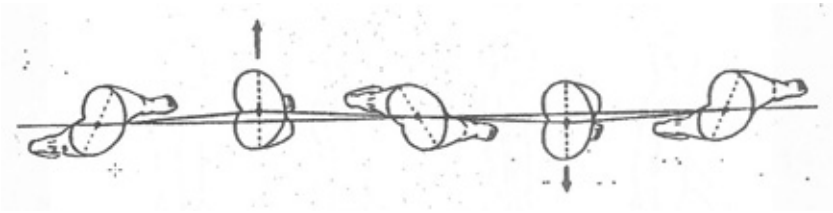


Figure III.10: Stride map and Lateral displacement of pelvis [41].

ground and the other leg starting to take off. In this stage the body can handle any speed or acceleration change in the treadmill. So detecting this phase in the algorithm is important because the treadmill is changed in this phase, chance of being imbalanced is much lower [40].

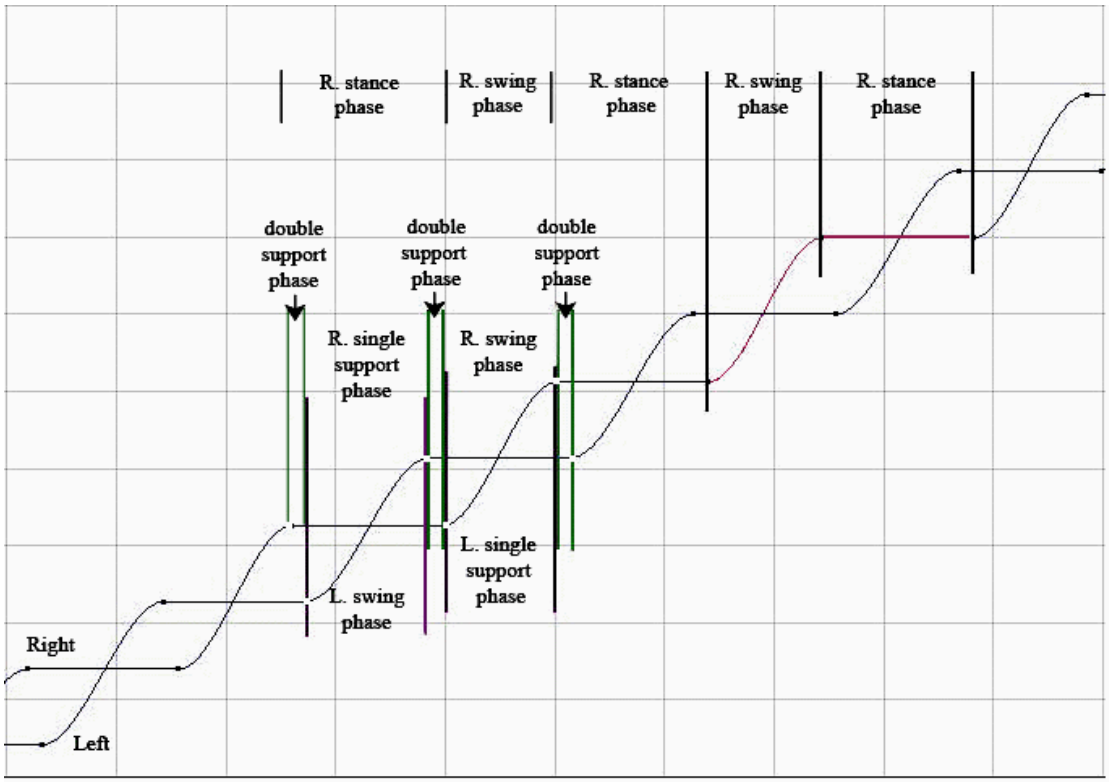


Figure III.11: Stride map and Lateral displacement of pelvis [41].

IV. METHODOLOGY

As mentioned before, CyberWalk platform is the only unconstrained omnidirectional treadmill described in [14, 20, 13]. The device uses costly *Vicon Motion Capture System* to determine the position of the head (headset contains three markers). The device weighs more than 10000KG and has an area of $4 \times 4m^2$ [13]. Aside from the mechanical limitations, the system also suffers from several issues:

- The system calculates 3D position from head pointers which is attached to the VR Headset. Using head as a tracking position creates problems as the head can move regardless of the body. So if a person leans forward or bends forward, the system may capture the movement as a speed component.
- How the system measures direction with respect to the head position is not described. The head can also rotate independent of the body. If the rotation is also controlled by head position, then the head must be kept straight.
- Uses marker based *Vicon Motion Capture* system, which is very precise but costly.

Here we present the methodology of our system, assuming an arbitrary *CyberWalk* like omnidirectional treadmill which takes velocity, acceleration and reference points as input values. But instead of using *Vicon Motion System* to calculate parameters, our setup uses Kinect camera to determine those parameters. Several approximation and design decisions have to be made before implementing the system. First, the system should support normal walking speeds, allowing users to walk in a natural motion and posture. The omnidirectional treadmill should not force the user to lean abnormally or apply extra force at

all times to walk. Second, it should be quick enough to change the speed and direction. Our method allows the user to move forward and backward with acceleration and deceleration rather than forcing the user to stay in the middle of the treadmill. This design decision was taken based on the work of [13, 14, 20]. According to the authors, when accelerating and decelerating, the vestibular system provides the brain with critical information concerning the changes in walking speed. This inertial input was shown to be important for the perception of walking speed and for maintaining postural stability. It turns out, the realization of the perception wasn't very easy to manipulate or replicate. That's why the users will be allowed to move forward or backward with acceleration or deceleration but slowly revert back to the original reference position by controlling the treadmill. Hence, speed of the treadmill gradually changes but at a lower acceleration rate than the user.

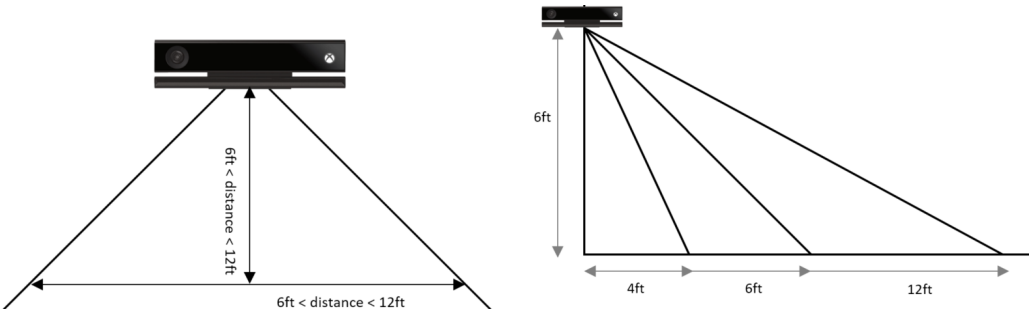
The algorithm can be used to control an omnidirectional treadmill within the limits of its size and speed to respond smoothly to changes in walking speed, allowing the user to start walking from standstill, to vary walking speed in a natural way, and even to abruptly stop walking without obtrusive changes in treadmill speed. The implementation of the algorithm was done using Microsoft Kinect, a low-cost RGBD capturing device as it is readily available in the market.

Before describing the algorithms, describing the assumed position and setup of the omnidirectional treadmill is important.

Proposed Device Position and Setup

The system is based on a treadmill of reasonable size deduced from empirical data. For best results, a size of $19.6 \times 19.6 ft^2 (6 \times 6 meter^2)$ is suggested in [13]. But due to space, weight and cost limitation, their design used only $13.2 \times 13.2 ft^2 (4 \times 4 meter^2)$ area. Other omnidirectional treadmills used less than $6.5 \times 6.5 ft^2 (2 \times 2 meter^2)$ of area such as [42, 17]. Kinect has a range of $1.64 ft (0.5 meter)$

to $14.76ft(4.5meter)$ and 70° horizontal FOV and 60° vertical FOV as mentioned in Table III.1. According to Microsoft, the optimal distance of a user from the Kinect camera is six feet and the width of the space should be at least $6ft(1.8m)$ and not more than $12ft(3.66m)$ wide [43]. Pictorial description of these conditions are shown in Fig. IV.1. So Based on the information and diagram shown in Fig. V.1, we are proposing that an area of $6\times6ft^2(1.83\times1.83m^2)$ or $8\times8\ ft^2(2.43\times2.43m^2)$ will work best with the Kinect sensor. Several authors



(a) (b)

Figure IV.1: Suggested distance and height of the Kinect sensor by Microsoft

suggested that placing the Kinect camera angled at 45° with respect to the user produces good motion data [37, 36]. They also suggested to keep the Kinect at a distance of almost $9ft$ and a height of $4ft$ as depicted in Fig. IV.2. While the

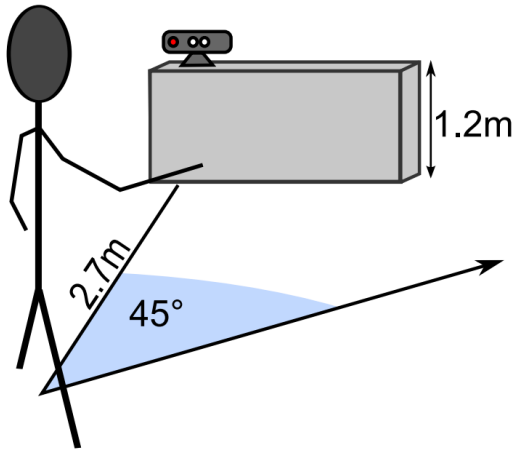


Figure IV.2: Distance and position of the Kinect sensor used by [37]

main idea is to avoid as much occlusion as possible [39], in our case, upper body occlusion or hand gesture is not a concern. Gathering occlusion free lower body

joints are the primary goal here. Hence, the angle of Kinect is chosen as 30°. An overall treadmill concept design based on the parameters described above and tailored in the fashion of *CyberWalk* [13], is shown in Fig. IV.3 from a top-down view.

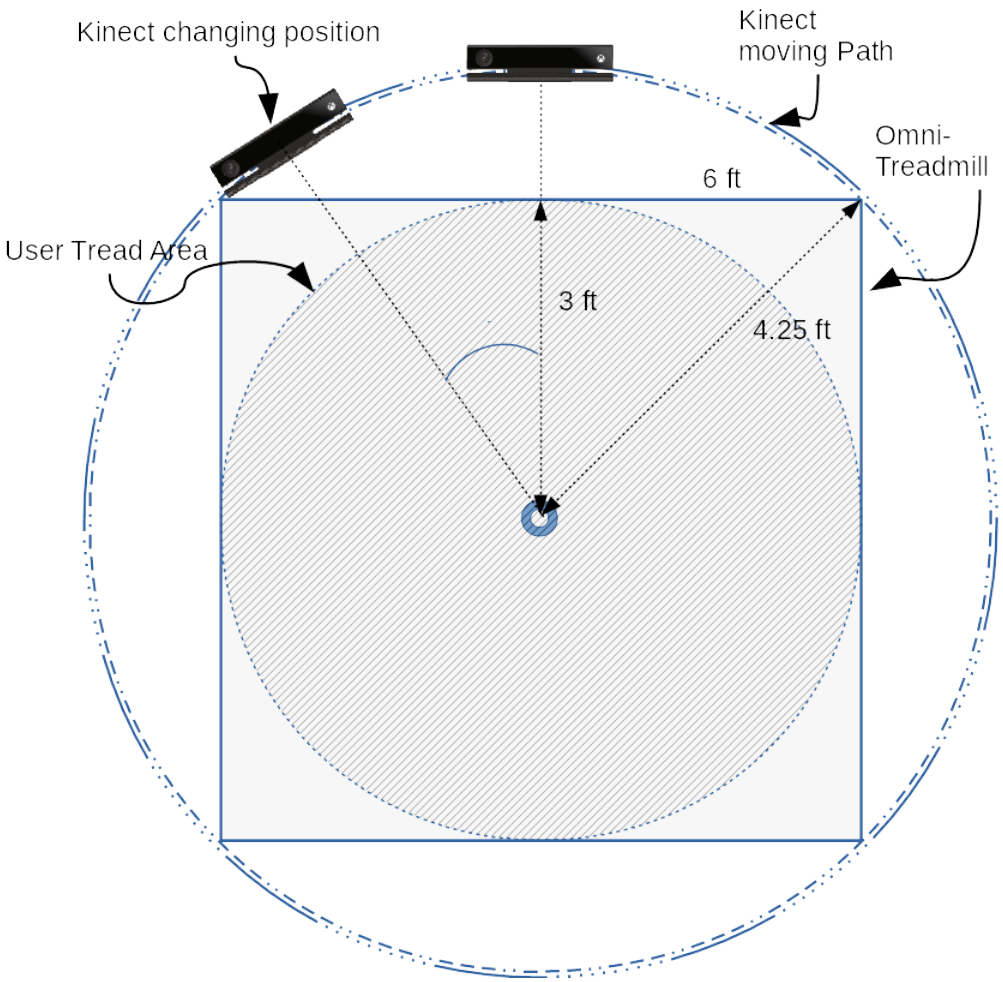


Figure IV.3: Proposed conceptual design of Omnidirectional Treadmill

Control Algorithm

Algorithm Constraints

The algorithm allows users to start walking on a stationary treadmill. Gradually, it will respond to the speed of users which is a constant input to the treadmill by this algorithm. Likewise, if the user stops walking, the treadmill eventually stops and brings the user back to the center of the treadmill. This strat-

egy works best as it doesn't involve manipulating the inertial input to the user which is important for the perception of natural walking speed and postural stability [13]. The normal walking cycle has three main phases; acceleration phase, deceleration phase and a steady or rhythmic phase [44]. During the steady state phase, walking is mainly dominated by vision as well as relative motion of the objects. During acceleration and deceleration phase, users can feel the change in speed of the treadmill. This is unavoidable because treadmills with bigger area allows for changes in treadmill speed that are low enough to maintain the postural stability of the user, but that also suffers from noticeable acceleration [13]. The maximum allowable walking speed of omnidirectional treadmill is 1.5 m/s. Acceleration up to 1 m/s^2 should be possible according to [13]. Hence, our algorithm will maintain these limits at all times.

Another set of constraints is the relative position of Kinect Camera when the user is changing direction. The Kinect has a FOV of about 60°; when users are changing direction, there can be possible occlusion which can hamper the operation of the algorithm. Hence, we chose to rotate the Kinect camera around the square treadmill in a circular fashion to sustain proper algorithm outputs. That is, the direction output of our algorithm will also rotate the Kinect so that the angle between Kinect and user is always zero. This is shown in Fig. IV.3.

Algorithm Flowchart

The basic steps of the algorithm are depicted in Fig. IV.4 and IV.5.

Joint Smoothing

The algorithm starts by capturing data from Kinect using our implemented software which is responsible for data capture and real-time calculations. After joint data is captured, the Spine-Base, Left feet and Right feet are tracked by the

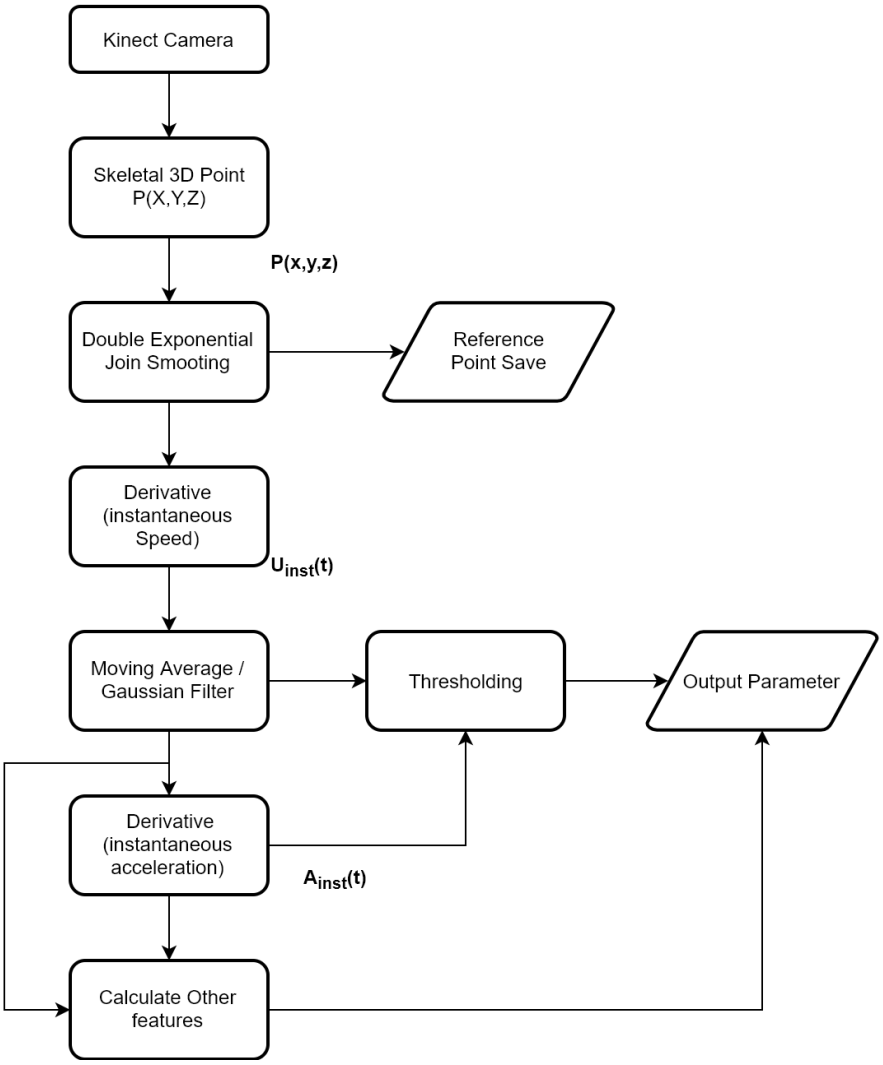


Figure IV.4: Basic Steps of the Algorithm

software as suggested in Chapter III. As each joint data generated noise from the environment and small body movements, it needs to be smoothed. But as the data have to be processed realtime, exponential smoothing is used. Exponential smoothing is originally a weighted average of the past observations, with exponentially decaying weights as the observation get older. Simply put, recent observations have a higher associated weight than the older ones. Basic equation for exponential smoothing is shown in (IV.1) and (IV.2).

$$S_{t+1} = \alpha x_t + (1-\alpha) S_t, \qquad 0 < \alpha \leq 1, t > 0 \qquad \text{(IV.1)}$$

In other words, the new value is the old value plus an adjustment for the error that occurred in the last adjustment i. e.

$$S_{t+1} = S_t + \alpha\epsilon_t \tag{IV.2}$$

Where ϵ_t is the prediction error, S_t is the smoothed data at t and α is the smoothing parameter. But single exponential suffers from trending values. And our position data should have a data trend where the z-axis data will have a sinusoidal-like shape. Hence, double-exponential smoothing is used by introducing second constant γ. The equations are shown in (IV.3):

$$S_t = \alpha x_t + (1-\alpha)(S_{t-1} + b_{t-1}) \qquad 0 \leq \alpha \leq 1 \tag{IV.3}$$

$$b_t = \gamma(S_t - S_{t-1}) + (1-\gamma)b_{t-1} \qquad 0 \leq \gamma \leq 1 \tag{IV.4}$$

Here, γ is the correction parameter, b_t is the current trend parameter at t. Besides double-exponential smoothing, we've also introduced jitter reduction filter. That is, if the position data deviates more than a certain amount (jitter radius, r_j) from the previous position value, it is calculated with another exponential like function shown in (IV.5):

$$S'_t = S_t \times \beta + S_{t-1} \times (1-\beta) \quad where, \beta = \frac{(S_t - S_{t-1})}{r_j}, 0 \leq r_j \leq 1 \tag{IV.5}$$

Reference Point and Floor Parameters

The smoothed data is then saved and can be used for floor-plane parameter calculations and reference setting. Kinect inherently provides the floor parameters by default, but for better calibration of floor parameters to reference feet positions, floor-plane parameters can be calculated from multiple points. At first, the user is asked to walk several times in one direction. After that, the data is pre-processed to find proper floor-parameters. The floor is defined by a

plane $ax + by + cz + d = 0$. If we have multiple points and have 4 variables (namely a, b, c, d), we set $c = -1$ for simplicity. Then replacing x, y with multiple points, we can solve for the floor parameters using Least-Square method shown in (IV.6) and (IV.8):

$$ax + by + d = z, \qquad assuming\ c = -1 \tag{IV.6}$$

$$\begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_3 & 1 \\ & \dots & \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ d \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \tag{IV.7}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = (A^T A)^{-1} A^T B, where\ x = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_3 & 1 \\ & \dots & \\ x_n & y_n & 1 \end{bmatrix},\ and\ B = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} \tag{IV.8}$$

After floor parameters are found, one can simply check whether a point is on the floor by putting x, y, z values in the equation, and if that is zero (or close to zero), then the point is on the floor.

Speed and Acceleration

Speed and acceleration calculations are straight forward. The elapsed time is calculated using a timer. Hence, the difference between a frame and the next frame is calculated and is called elapsed time t. The instantaneous speed of motion for desired skeletal joint is calculated as the resultant of x, y, and z positions over subsequent frames that represent a motion. Equation (IV.9) repre-

sents speed equation used in the algorithm.

$$U_{inst}(n) = \frac{\mathrm{d}}{\mathrm{dt}}(x, y, z)\Big|_{T=nt} = \frac{1}{t}\sqrt{(x_n - x_{n-1})^2 + (y_n - y_{n-1})^2 + (z_n - z_{n-1})^2} \tag{IV.9}$$

It might also be necessary to only calculate the Z-axis component. In that case derivative of Z component is used. In a similar fashion, acceleration is calculated from derivative of speed, referring to (IV.10):

$$A_{inst}(n) = \frac{\mathrm{d}}{\mathrm{dt}}(U_{inst})\Big|_{T=nt} = \frac{1}{t}(V_{inst(n)} - V_{inst(n-1)}) \tag{IV.10}$$

However, after speed calculation but before acceleration, the speed values need to be filtered, as Kinect coordinates have jitter and other irregular components, even after applying exponential smoothing. Therefore, a smoothing needs to be applied to the calculated speed values so that values passed to omnidirectional treadmill are smooth and don't contain abrupt changes. Based on trial and empirical data, triangular smoothing was chosen. The triangular smoothing is similar to the weighted rectangular smooth. (IV.11) Represents a 5-point triangular smoothing:

$$S_t = \frac{x_{t-2} + 2x_{t-1} + 3x_t + 2x_{t-1} + x_{t-2}}{9} \tag{IV.11}$$

Triangular smoothing doesn't have any significant effect and preserves the area under where it is a straight line. For implementation purpose, two passes of four point rectangular smoothing was chosen. The general rule of thumb is, n passes of a w-width smooth results in a combined width of $n \times w - n + 1$. Therefore, two passes of four point rectangular smoothing will result in a 7-point triangular smoothing.

Data Output Filtering

Even though speed and acceleration are calculated per frame, they are not passed to omnidirectional treadmills per frame. Based on several logic testing as shown in Fig. IV.5, the values are passed to omnidirectional treadmills. The first condition is checking the movement of front feet. It can be left or right feet, but the front feet will be tracked. As described by the authors Xu et al. in [28], the back feet suffers from an error as it is sometimes occluded from Kinect. The front feet on ground test is based on the floor parameters calculated in (IV.6) to (IV.8). If multiplying the correspondent feet position with these floor parameters results in a value within a threshold value (ideally zero), then the feet are on the floor. Then step length is calculated as the distance between the Z-axis of feet positions, which can be optional. The rotation vector is then calculated for $X - Z$-plane in reference to Kinect position.

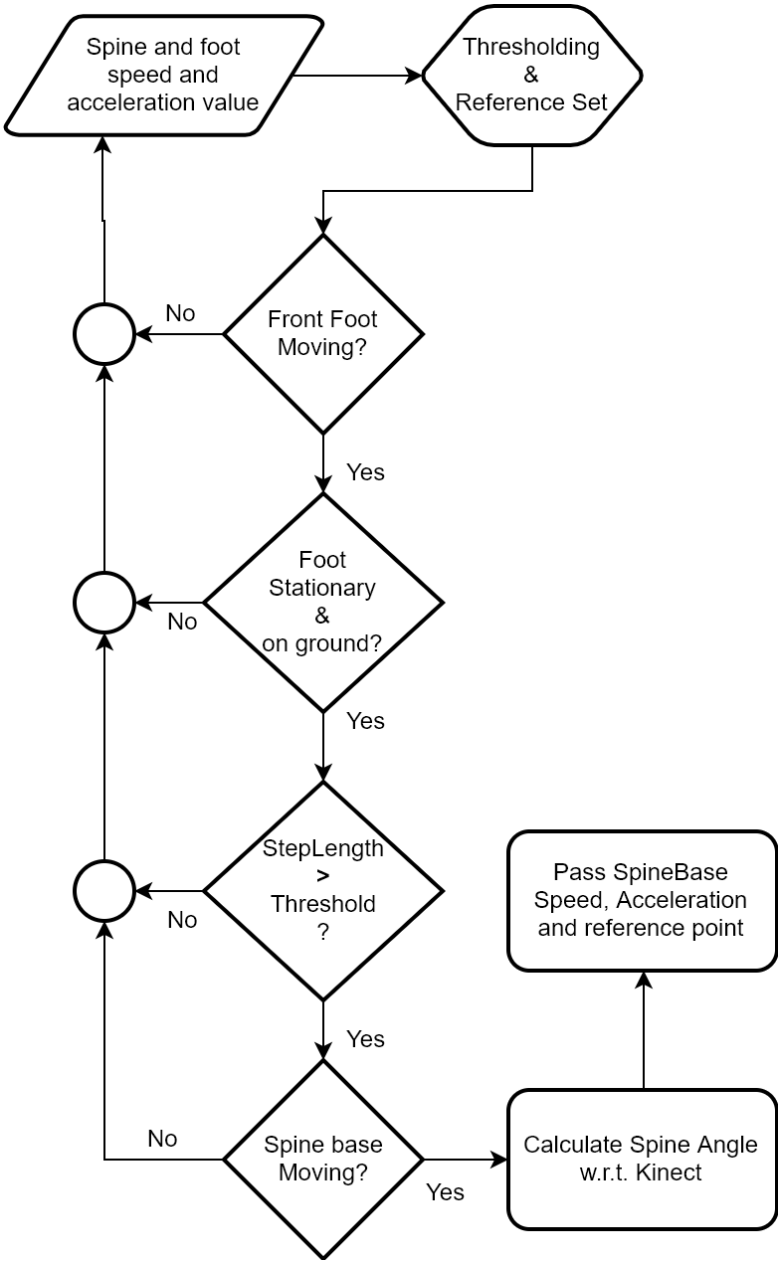


Figure IV.5: Determination of Data output

V. EXPERIMENTS AND RESULTS

Testing Environment

All testings was performed using a PC with configuration below:

- Kinect for XBox One (v2)
- Intel Core i7-4720HQ 2.6GHz Quad-Core
- 8 GB DDR3 RAM
- Microsoft Windows
- Visual Studio 2015 for Development
- Matlab for Analysis

The Kinect is placed straight in front of the user as seen in Fig. IV.1a. To see the lower body joints clearly, being perpendicular to them is the ideal case, which is not possible as Kinect can't measure only the lower body joints; it has to measure the whole body joints. In light of that, the Kinect is placed at a height on a head level, we chose 6ft as shown in Fig. V.1.

Another important factor is the angle in which the Kinect sensor is tilted, as we want to cover the average height of a human ($\sim 6ft$). As shown in Fig. V.1, choosing an angle of 0°, the minimum distance to the user is $10.39ft$ ($3.17meter$). Any lower distance, the Kinect will lose track of the user's lower body. Choosing this angle, makes the available tread area lesser as maximum range for Kinect is $\sim 14.76ft$ or $4.5meter$. Hence, 30° of angle is chosen where the area of operation is sufficient and the height of an average human is supported.

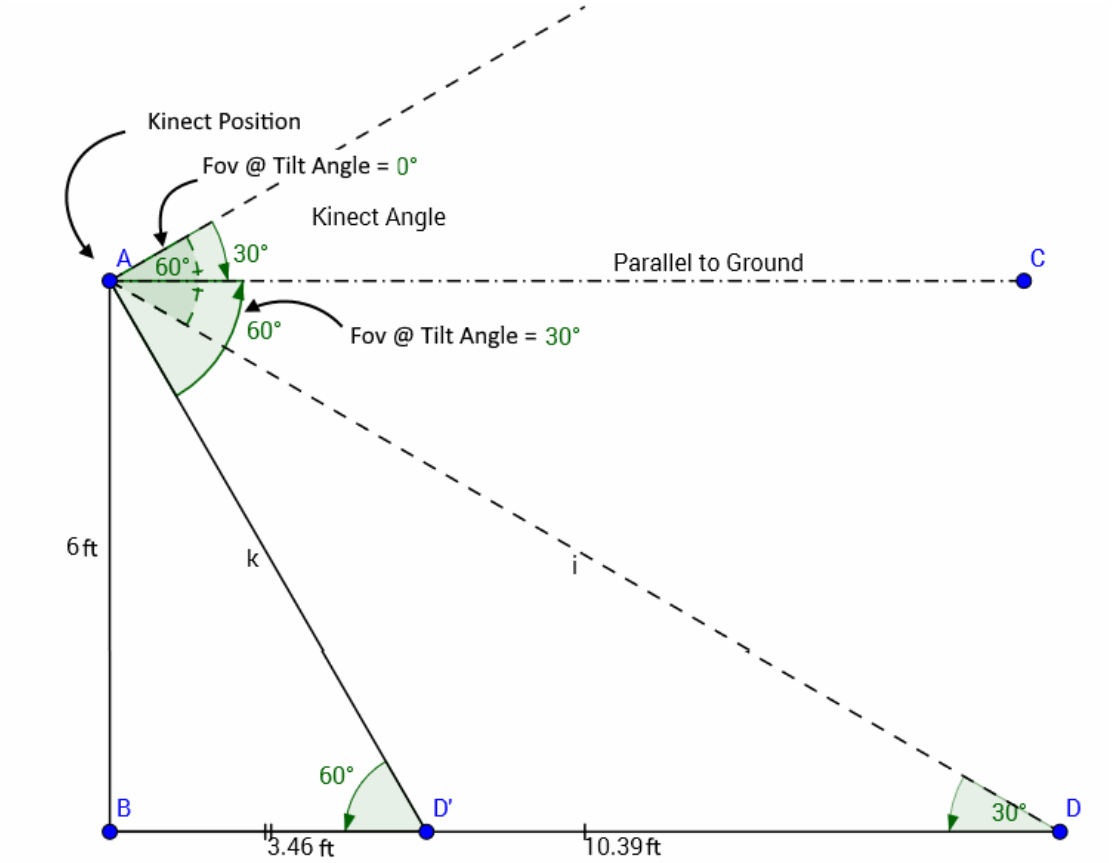


Figure V.1: Proposed tilt angle and distance of the Kinect Sensor

Testing Methods

As omnidirectional treadmill was not available at the time of experiments, participants were asked to walk starting from a distance of $12ft(3.66\ m)$ towards the Kinect Sensor for acceleration and speed calculation. On a second run, they were asked to walk towards several angular markers ranging from $-20°$ to $20°$ to measure the angle, speed and acceleration. The participants were directly facing the Kinect sensor at first, behind which a screen displayed the running program with a video capture of their movements and rendered skeleton for their reference. In each of these experiments, joint positions were recorded for all available joints for each frame with no frame-gap. The speed, acceleration and rotation values were calculated realtime. Furthermore, position data, calculated speed, angle and acceleration values were saved for further analysis.

After reading Kinect joint positions, three joints are tracked; the spine-base, left foot and right foot. After that, exponential smoothing is applied. As the Kinect joint values tend to be jittery, an exponential smoothing with smoothing factor of 0.25, correction factor of 0.25, prediction factor of 0.25 and jitter radius of 0.05 is used. The Fig. V.2 below represents the result of joint positions with smoothing and without smoothing: Although, the effect is not much visible in

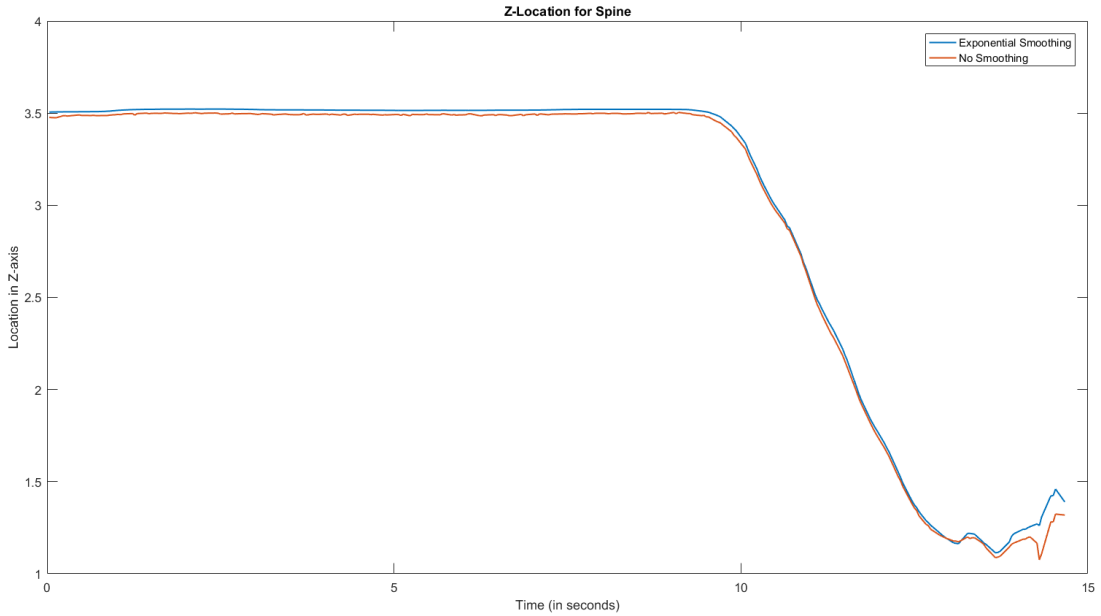


Figure V.2: Z Location data for Spine using exponential smoothing.

the figure, it is quite visible after calculation of speed, as shown in Fig. V.3: The correlation coefficient calculated between the original and exponential have an average of 0.8856 with range $[0.9572, 0.7426]$. Hence, the signals are almost similar to each other.

Speed and acceleration calculations are done in realtime. But before that, choosing one smoothing technique was necessary. The depiction of speed calculation using three different methods are shown in Fig. V.4, V.5 and V.6 Among these smoothing techniques, the Gaussian and Triangular works better as we expect the speed to smoothly increase and decrease. Also, too much deviation from original data is avoided. As smoothing is a filter, it imposes delay

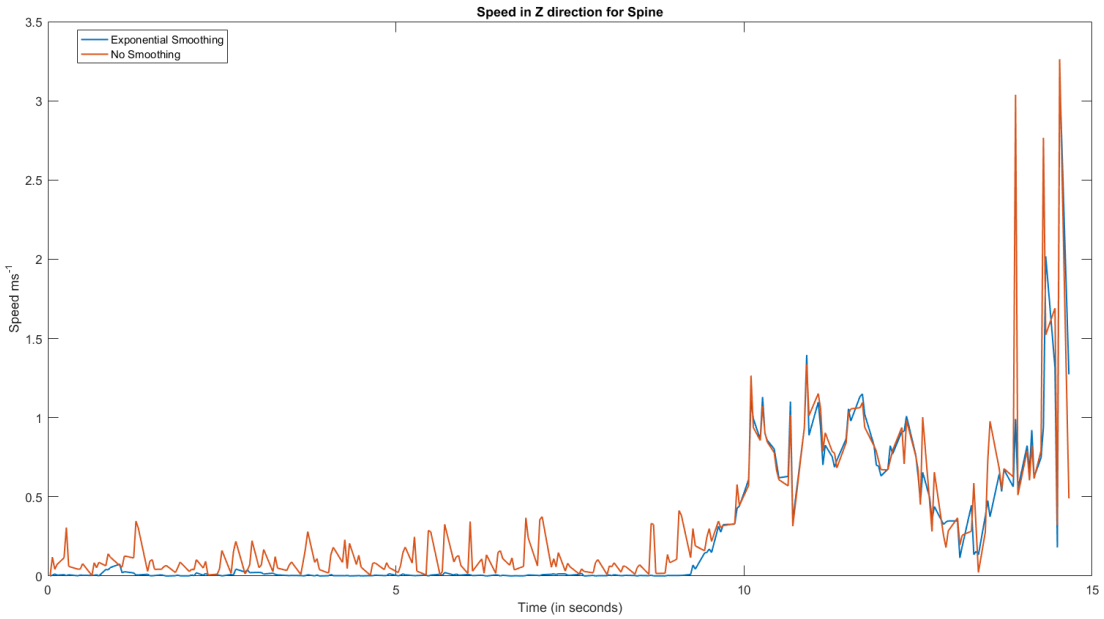


Figure V.3: Result of Spine Speed in Z direction after using exponential smoothing

based on window size. Hence, a window size of seven is chosen so that, the delay remains minimum and the speed data can be smoothed to our need. This deviation is determined from correlation value and percent Mean-Squared-Error (MSE) value. Table V.1, V.2 and V.3 below shows the correlation and MSE of all three joints over several data sets for different smoothing methods. It can

Table V.1: Correlation and MSE for different smoothing technique for Spine

Gaussian		**Triangular**		**Moving Average**	
Corr	**MSE**	**Corr**	**MSE**	**Corr**	**MSE**
0.87147	2.693%	0.82279	3.625%	0.7827	4.309%

Table V.2: Correlation and MSE for different smoothing technique for Left Foot

Gaussian		**Triangular**		**Moving Average**	
Corr	**MSE**	**Corr**	**MSE**	**Corr**	**MSE**
0.90185	4.3024%	0.85167	6.368%	0.77169	9.235%

be seen from the correlation and MSE values that the Gaussian smoothed value is more suitable and correlated to the original value while retaining important data. Hence, Gaussian was chosen as the default smoothing filter. Another point can be noted that, the spine-base joint has the least error and largest correla-

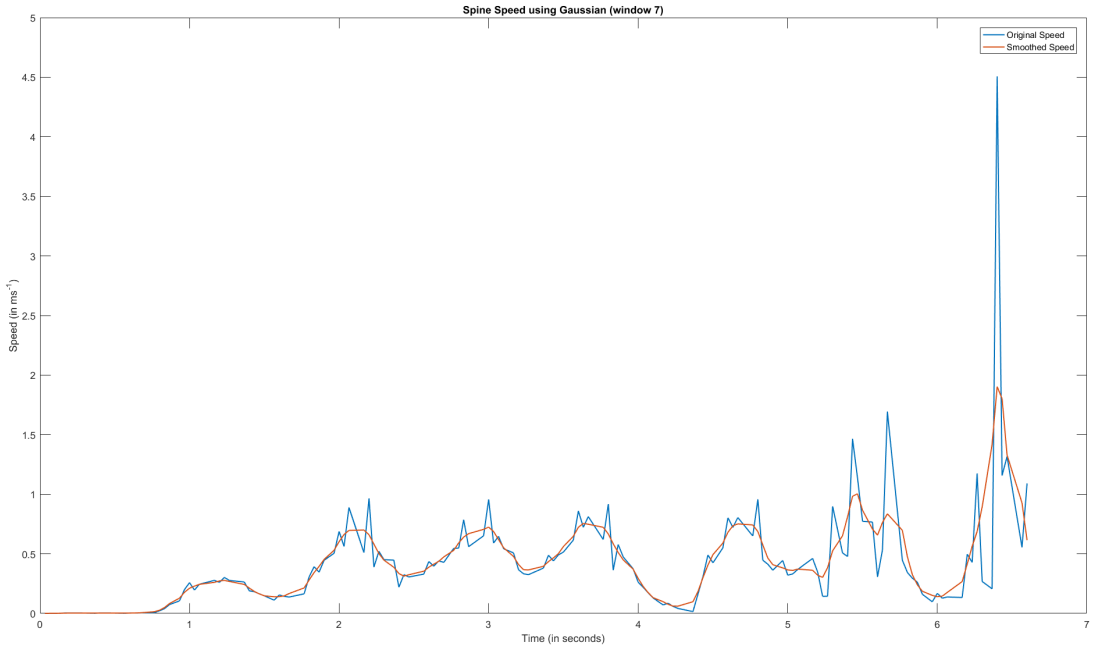


Figure V.4: Spine Speed using Gaussian (window size seven)

Table V.3: Correlation and MSE for different smoothing technique for Right Foot

Gaussian		**Triangular**		**Moving Average**	
Corr	**MSE**	**Corr**	**MSE**	**Corr**	**MSE**
0.8953	6.096%	0.84066	8.872%	0.75637	12.806%

tion w.r.t other joints. As a result, we can conclude that our original hypothesis of selecting the spine-base for determining omnidirectional treadmill speed and acceleration value was correct. Also spine-base position is pretty immune to tilting, rotating and leaning the body. The data output algorithm shown in Fig. IV.5 also rules out a several non-walking conditions which makes this control system superior to *CyberWalk* system. Hence, our control algorithm solves the first problem of *CyberWalk* system described in Chapter IV. The acceleration is calculated for each of the three joints in realtime using (IV.10). Fig. V.7a, V.9a and V.8a below are the plot of acceleration values which are derivatives of speed. We can see, the acceleration values have noisy components which needs to be further smoothed. Smoothing the acceleration values are important. Based on the acceleration values, the omnidirectional treadmill will change the rate of velocity increase or decrease. If the values are changed frequently, the omnidi-

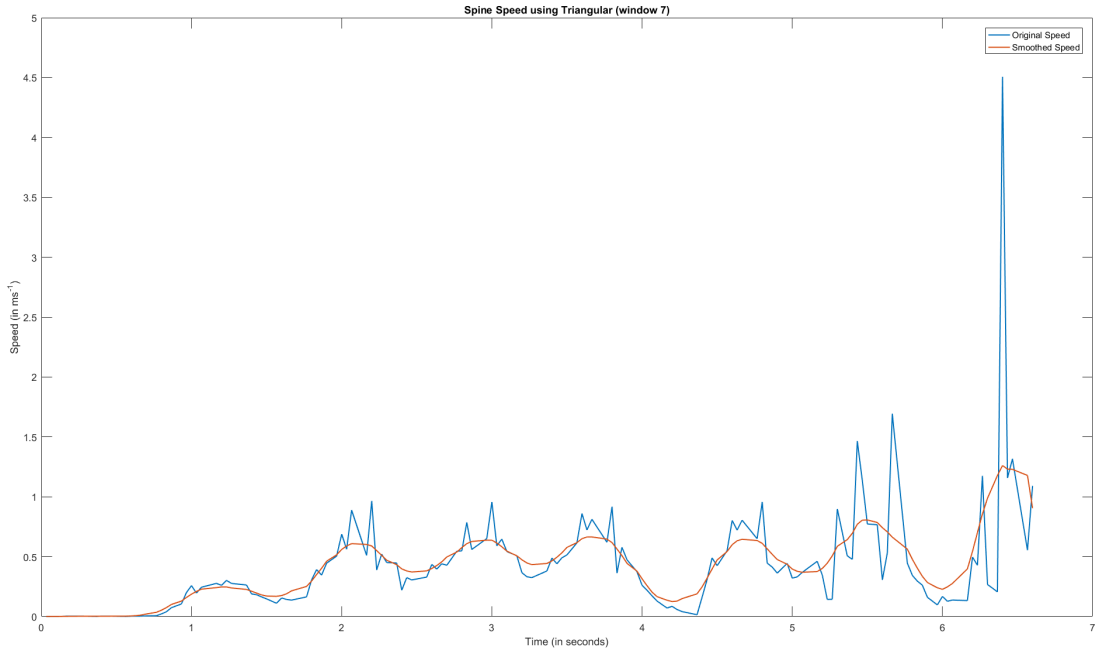


Figure V.5: Spine Speed using Triangular Smoothing (window size seven)

rectional treadmill cannot cope up with that, and even if it can, it may cause the user to fall out of balance. Hence, smoothing the acceleration values are also important.

The angle is calculated simply as a normal component to the Spine-Base point with respect to the hip bones. As a normal component it represents a vector direction. Then an angle is calculated with this normal component and the vector from spine-base to the Kinect sensor. It may also be easy to use Kinect Supplied radian-angle around y-axis. We can get the cumulative angle w.r.t the Kinect camera which is the center of camera 3D space. After getting the cumulative angle, we have to smooth the data to make it usable. A simple experiment with angular change was done. The subject was told to change direction on every step. Fig. V.10 shows angular value w.r.t acceleration for that experiment. Notice that, angular values sways from a fixed position. As shown in chapter III, Fig. III.10, there is a lateral displacement in both angle and position. The direction changing angular values represent the displacements.

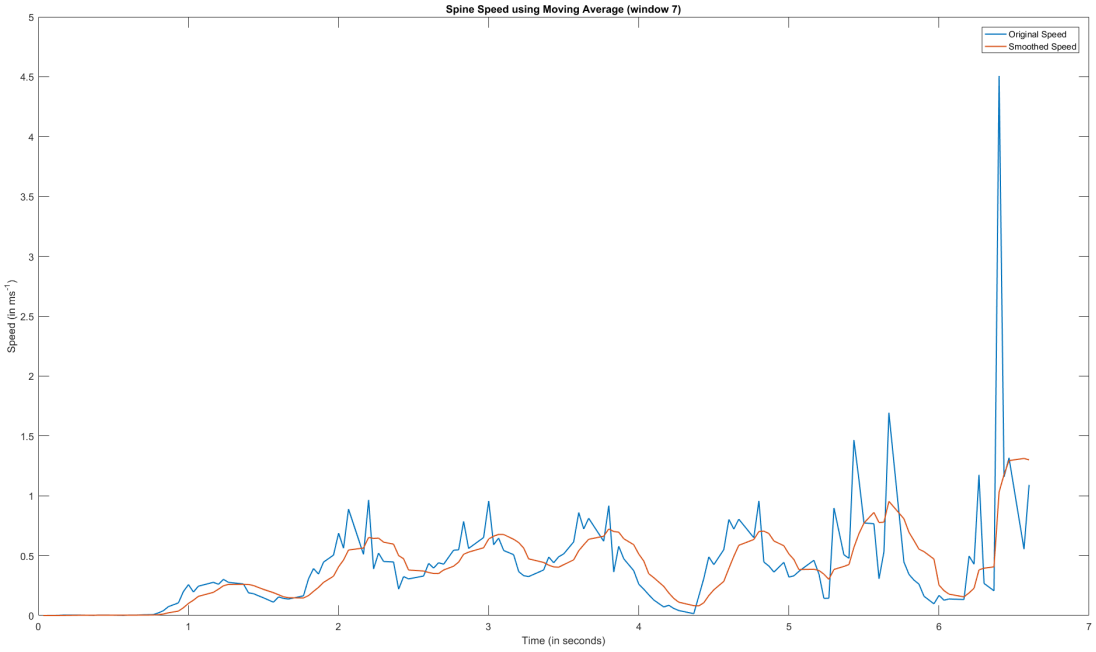
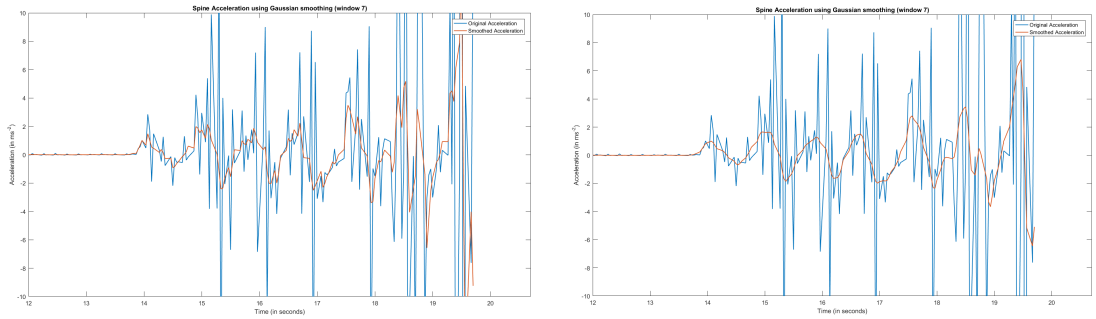


Figure V.6: Spine Speed using Moving Average (window size seven)

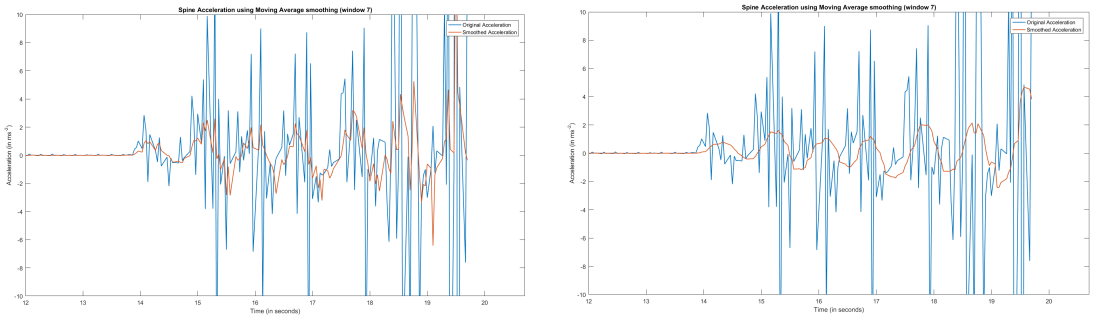


(a) (b)

Figure V.7: Spine Acceleration using no smoothing and Gaussian smoothing(window size seven)

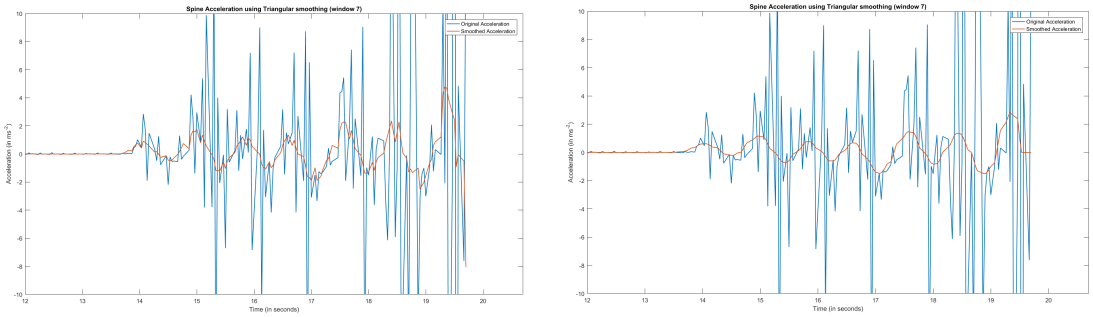
Validation

Two separate experiments were conducted to validate the speed and angular calculations. As an omnidirectional treadmill was not available, to validate the algorithm's speed and angle calculation, these experiments were necessary. First experiment, for validation of speed calculation by the algorithm, the user was told to walk in the same speed for a fixed distance multiple times (multiple runs). The Kinect has a range of $4.26m(14\ ft)$ and the proposed treadmill area was $1.83 \times 1.83m^2(6 \times 6ft^2)$ or $2.43 \times 2.43m^2(8 \times 8\ ft^2)$. For this reason,



(a) (b)

Figure V.8: Spine Acceleration using no smoothing and Moving Average smoothing(window size seven)



(a) (b)

Figure V.9: Spine Acceleration using no smoothing and Triangular smoothing(window size seven)

we've chosen to cover distance of $2.29m(7.5\ ft)$ for this experiment, starting from $3.66m(12\ ft)$ and ending in $1.37m(4.5\ ft)$. So, the user walked distance of $2.29m(7.5\ ft)$ in a fixed speed multiple times for each time range. Total range of speed calculation $0.2ms^{-1}$ to $1.4ms^{-1}$. Time in milliseconds was noted using stopwatch, and average speed was calculated for the chosen distance. Kinect data was also acquired for speed calculation and comparison. There were six ranges of speed tested. The speed ranges were categorized based on time taken to achieve that speed. Time ranges measurements were '1 to 2 seconds', '2 to 3', '3 to 4', '4 to 5', '5 to 6' and '6 to 7 seconds'. For each time category, 10 walk samples were taken. Table V.4 below shows the average time measured for $7.5\ ft$. Based on this time measurement, the measured speed and Kinect calculated speed is also shown in Table V.4.

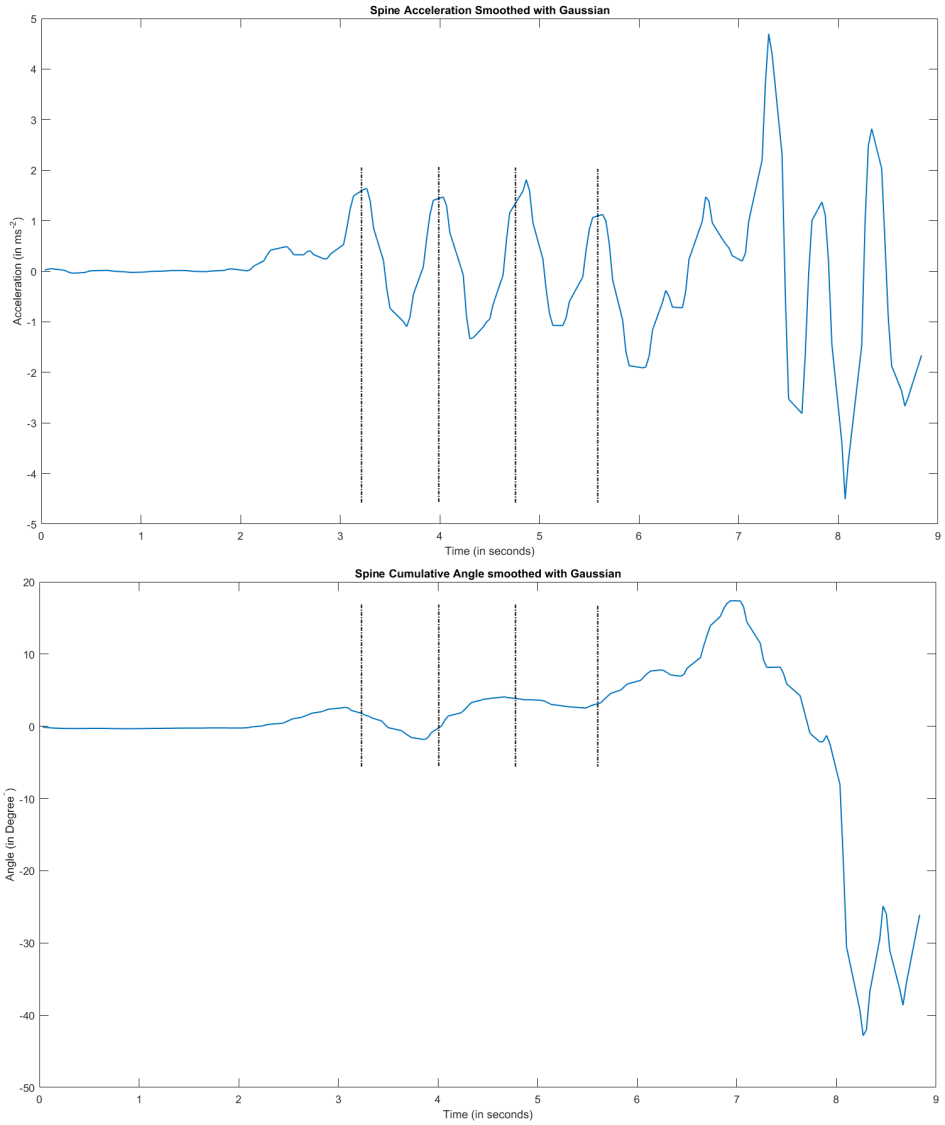


Figure V.10: Change of angle with respect to change of acceleration

According to Table V.4, it is apparent that Kinect's speed calculation error is increasing for higher speed. Speed calculated by the Kinect had an average error rate of 18.91% for $> 1m/s$ speed.The equation for percent error calculation is shown in (V.1).

$$\%Error = \left| \frac{Theoritical\ Value - Experiment\ Value}{Theoritical\ Value} \right| \times 100 \tag{V.1}$$

Other speed ranges had low error rates than the high speed one. The average error of all the speed(except $> 1m/s$) is 8.21%. Hence, it can be concluded that the Kinect can handle upto $1m/s$ speed with reasonable error margin ($< 10\%$)

Table V.4: Speed Comparison Table for distance of $7.5ft$ or $2.286m$. Speed Unit m/s, Time is in seconds

Range(seconds)	**1 to 2**	**2 ot 3**	**3 to 4**	**4 to 5**	**5 to 6**	**6 to 7**
Time measured	1.9330	2.4315	3.3428	4.4667	5.4021	6.6086
Speed(Measured)	1.1843	0.9411	0.6855	0.5131	0.4247	0.3468
Speed(Kinect)	0.9597	0.8370	0.6380	0.4650	0.3908	0.3150
%Error	**18.91**	**10.97**	**7.06**	**9.45**	**8.03**	**9.23**

in this setup. The overall error rate including all speed ranges is 10.12%. Table V.4 represents the average of all speeds taken during the experiment setup. A detailed listing of timing and speed calculations for multiple sample can be found in Table A.1, A.2 in Appendix A. Also, speed calculations for Kinect and it's percent error with respect to measured speeds can be found in Table A.3 and Table A.4 in Appendix A.

The second experiment, validates the accuracy of angular measurements. The experiment setup is shown in Fig. V.11. The dotted lines represent the angle markers, which were taped in the ground. From -20 to 20 degrees, each marker was placed after five degrees. In this experiment, the subject is asked to face towards each angle-marker multiple times for three to five seconds. Based on this experiment, the angles were calculated based on Kinects data. Table V.5 shows the calculated angle values by the algorithm, their percentage error and the standard deviation.

As mentioned, for each angle multiple samples were taken. The Kinect angle mean represents the average of all of the samples. 'Abs Diff' represents the difference between the actual angle and the calculated mean angle. The percent error was calculated using (V.1). The 'STD' of each angle is the standard deviation of all the angles calculated for that specific angle. The 'Max' and 'Min' angles are self-explanatory. It is apparent from Table V.5 that, $\pm 20°$ has the highest percent error ($> 10\%$) and a displacement of more than $\pm 3°$. Although, $15°$ has an error rate of $< 10\%$, from the standard deviation, it is apparent that the

Table V.5: Angle Measurement validation. Unit of angles are in Degree(°)

Original	Kinect Angle					
Angle	Angle(Mean)	Abs Diff	%Error	STD	Max	Min
0	0.0452	0.0452	4.5197	0.8267	1.85	-1.02
5	5.0459	0.0459	0.918	0.8114	6.85	3.39
10	9.2784	0.7216	7.216	0.9799	11.28	8.13
15	13.466	1.534	10.2267	1.658	14.97	12.14
20	16.146	3.854	19.27	3.9012	17.51	14.81
-5	-5.3183	0.3183	6.366	0.5522	-4.43	-6.6
-10	-10.983	0.983	9.83	1.0135	-9.11	-12.79
-15	-13.514	1.486	9.907	1.3253	-12.25	-17.22
-20	-17.353	2.647	13.235	2.8504	-16.12	-19.90

angle calculation varies a lot within 1.5σ. The rest of the angles, namely, $\pm 0°$, $\pm 5°$, $\pm 10°$ has a lower error rate than $\pm 15°$ and $\pm 20°$. Also their standard deviation values suggest that, the tendency of varying their values are lower. Hence, it can be deduced from Table V.5 that, the best angle range for this setup and algorithm is $-15° < Angle < 15°$.

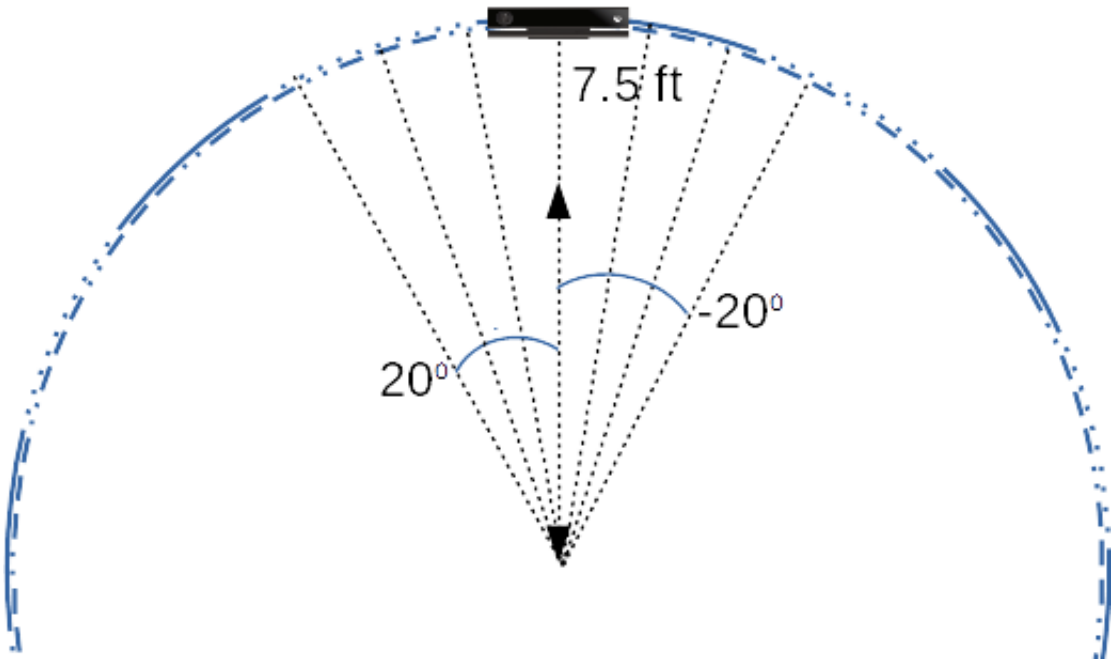


Figure V.11: Angle validation experiment setup

The software was implemented in .NET, Microsoft Visual Studio. Several sample run of the software are shown in Fig. V.12a to V.12e. The basic class diagram of

(a) (b) (c) (d) (e) (f)

Figure V.12: (a) is the software setting reference and floor parameter. (b) to (f) shows the working condition of the software when a user walks toward the Kinect and in an angle

the software is shown in Fig. V.13.

The main class developed was $MainWindow$, responsible for reading data from a Kinect device. Then it creates $BodyEx$ and $JointInfo$ class objects which actually are a placeholder for all body joints data and joint positions

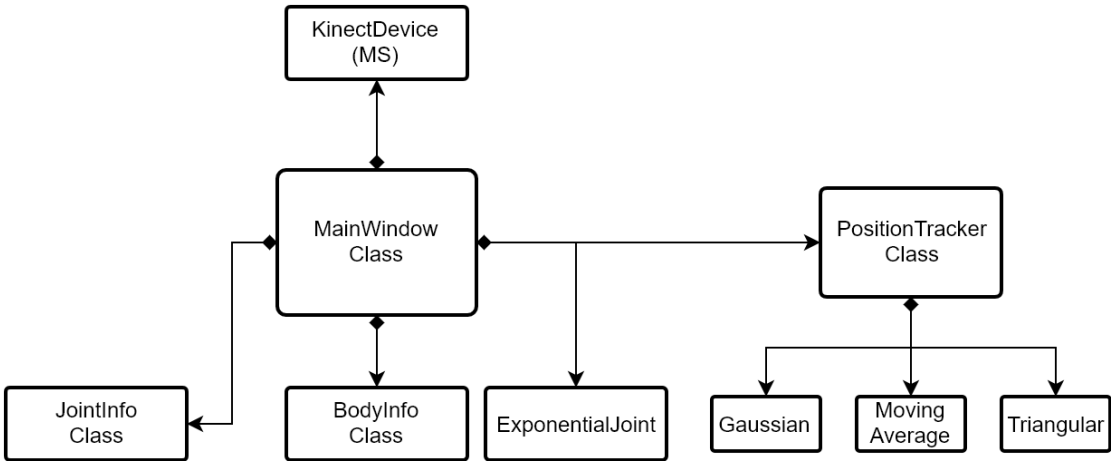


Figure V.13: Basic Class diagram of the Software Used

and speed. The *PositionTracker* class does the actual work of smoothing the data and calculating acceleration and speed in realtime. *ExponentialJoint* class features the exponential smoothing of the joint positions. Core code segments from the software are listed in appendix VI. The average lag for the processing of these data and smoothing without optimization was $\sim 3ms$. As optimization wasn't the priority here, details like using multi-threading or Graphics Processing Unit (GPU) inclusion wasn't implemented, which could make the software run faster.

VI. CONCLUSIONS AND FUTURE WORK

Conclusions

The challenge of developing a control algorithm for omnidirectional treadmills is largely correlated to the actual design of the omnidirectional treadmill. Independently designing the control algorithm poses several limitations. As the speed, acceleration and rotation values are calculated, they cannot be verified against an actual output of omnidirectional treadmills. Hence, existing motion and acceleration can be further parameterized to proper values required for omnidirectional treadmills. This research confronts three major problems of existing algorithms as described in Chapter IV. Our algorithm addresses the first problem by choosing the appropriate smoothing technique and then choosing spine-base as a reference for calculating velocity and acceleration. As mentioned before, the spine-base position values are much more accurate than the other joints counterpart. The second problem of rotation value was addressed in Experimental Section Chapter V.

Angular values were calculated and validated to be changing properly according to the acceleration value. Still, without the omnidirectional treadmill setup, there are very few and expensive way to measure the accuracy of angle, speed and acceleration. The most important problem was reducing cost of existing omnidirectional treadmill. As the omnidirectional treadmill itself tends to be costly, using a motion capture device for reference calculation increases the cost, although it provides accuracy. Instead of using a motion-capture camera, a Kinect Camera would be much more affordable and available, although sacrificing accuracy. It can still be used upto certain acceptable error margin($< 10\%$) as it is still used by many researchers for medical and multimedia purposes.

Acceleration and speed is calculated, but there is no straight-forward

way to calibrate the calculations, verify or benchmark the values. Although, for validation purpose only, a proper value change pattern is necessary, which we have as shown in Fig. V.7a, V.7b, V.9a and V.4. Also, two experiments were conducted to verify the standby angle and speed of the setup which is described in Chapter V. Based on the results, it can be said that this setup and algorithm works best for angle range of $-15° < Angle < 15°$ and a speed range of $.2ms^{-1} < Speed < 1.4ms^{-1}$.

Aside from calculating velocity, acceleration and rotation, the algorithm can also distinguish basic walking patterns. It can distinguish walking motion from non-walking motion, e. g., body lean, tilt or rotate. Walking is determined as a series of events, e. g., touching the floor, motion of a foot and spine, etc, which makes the algorithm susceptible to abnormal walking behavior. A user can lean forward to look at an object or rotate to watch another scene; our system will differentiate those and parameters won't be passed to the treadmill. This is another advantage of our proposed system.

Limitations

The limitations of our system are listed below:

- The biggest limitation of this system is the framerate of Kinect Camera. While the Kinect hardware is one of the most used research devices, it suffers severely from framerate-deficiency. Kinect's maximum framerate is $30Hz$, which is true for most of the cases. But even $30Hz$ framerate is not enough for realtime applications. Kinect have been used successfully for many medical analyses where researchers collect data from Kinect and post-process [35, 36, 40, 34, 28], but our system is a realtime system which needs to have good framerates. Average elapsed time for each frame was found to be $40.15ms$ with range $[33.33ms,\ 44.17ms]$. .
- While the temporal accuracy is good [28], Kinect acceleration and speed

values are noisy. As a result, smoothing is required. Even after that, spatial accuracy is not very reliable, which can be reduced by the subtraction of position values or increased [28]. Also realtime smoothing produces delay based on windowsize which can hamper the walking experience.

- The algorithm is not optimized enough to detect jumping, side-walking or cross-walking.
- It is assumed that after passing the acceleration and velocity the omnidirectional treadmill will automatically change the velocity over time based on acceleration. Instead, we can make the progression and give feedback to omnidirectional treadmills, to limit i's processing.

Future Works

Omnidirectional Treadmills will be the next generation of virtual reality exploring medium. This proposal will make the omnidirectional treadmill platform one step closer to being unconstrained and low cost. Future plans of this thesis are listed below:

- Implement the algorithm with an existing omnidirectional treadmill to verify it's functionality
- Apply a new validation method using Smartphone sensors. For example, Android and iOS smartphone sensors are very accurate, e. g., accelerometer, gyroscope, rotation vector, etc. We can make an app which can communicate with the software for synchronization, then capture gyroscope and accelerometer data. From the captured data it is possible to validate our calculated speed, rotation and acceleration with reasonable accuracy.
- Employ more test cases based on age, height and gender.
- Incorporate high framerate industrial cameras to detect depth from disparity and calculate position based on optical flow

- Apply deep-learning based human pose detection, which can be applied to our original proposal.
- Make a low-cost belt for the torso that will include a Smartphone sensor which will gather data and communicate with omnidirectional treadmill and VR headset rather than using camera to detect pose and position.

APPENDIX SECTION

APPENDIX A

Table A.1: Time measured for multiple runs

Range(seconds)	**1 to 2**	**2 ot 3**	**3 to 4**	**4 to 5**	**5 to 6**	**6 to 7**
Run 1	1.952	2.375	3.366	4.496	5.985	7.156
Run 2	1.765	2.398	3.428	4.483	5.437	6.707
Run 3	1.933	2.559	3.421	4.073	5.352	6.898
Run 4	1.956	2.523	3.351	4.779	5.594	6.274
Run 5	1.985	2.358	3.002	4.665	4.944	6.426
Run 6	2.011	2.376	3.489	4.304	5.101	6.191
Average Time	**1.933**	**2.4315**	**3.3428**	**4.4667**	**5.4021**	**6.6086**

Table A.2: Measured speed using time from TableA.1

Range(seconds)	**1 to 2**	**2 ot 3**	**3 to 4**	**4 to 5**	**5 to 6**	**6 to 7**
Run 1	1.1711	0.9625	0.6791	0.5084	0.3819	0.319
Run 2	1.2951	0.9532	0.6668	0.5099	0.4204	0.340
Run 3	1.1826	0.8933	0.6682	0.5612	0.4271	0.331
Run 4	1.1687	0.9060	0.6821	0.4783	0.4086	0.364
Run 5	1.1516	0.9694	0.7614	0.4900	0.4623	0.355
Run 6	1.1367	0.9621	0.6552	0.5311	0.4481	0.369
Average Speed	**1.1843**	**0.9411**	**0.6855**	**0.5131**	**0.4247**	**0.3468**

Table A.3: Calculated speed using data from Kinect

Range(seconds)	**1 to 2**	**2 ot 3**	**3 to 4**	**4 to 5**	**5 to 6**	**6 to 7**
Run 1	0.9309	0.8128	0.5577	0.4847	0.3419	0.2791
Run 2	1.0228	0.8440	0.6454	0.4308	0.3878	0.2988
Run 3	0.9503	0.8148	0.5988	0.5278	0.3920	0.3148
Run 4	0.9938	0.8675	0.6860	0.4385	0.3812	0.3361
Run 5	0.8631	0.8309	0.6980	0.4265	0.4129	0.3286
Run 6	0.9971	0.8514	0.6421	0.4815	0.4287	0.3320
Average Kinect Speed	**0.9597**	**0.837**	**0.638**	**0.465**	**0.3908**	**0.315**

Table A.4: Percentage error between Kinect's calculated speed from TableA.3 and measured speed data from Table A.2

Range(seconds)	**1 to 2**	**2 ot 3**	**3 to 4**	**4 to 5**	**5 to 6**	**6 to 7**
Run 1	20.51	15.55	17.87	4.67	10.47	12.63
Run 2	21.03	11.46	3.22	15.50	7.76	12.32
Run 3	19.64	8.78	10.38	5.95	8.22	5.00
Run 4	14.96	4.25	0.56	8.32	6.71	7.73
Run 5	25.05	14.29	8.33	12.94	10.69	7.61
Run 6	12.28	11.50	1.99	9.34	4.33	10.07
Average %Error	**18.91**	**10.97**	**7.06**	**9.45**	**8.03**	**9.23**

APPENDIX B

Software Core Codes

Code for Frame Processing

```
private void FrameProcess(object sender,
    BodyFrameArrivedEventArgs e)
{
  bool dataReceived = false;
  Vector4 floorPlane = new Vector4();
  double elapsedTime = 0;
  using (BodyFrame bodyFrame = e.FrameReference.AcquireFrame())
  {
        if (bodyFrame != null)
```

```
        {
            if (this.bodies == null)
            {
                this.bodies = new Body[bodyFrame.BodyCount];
            }
            // The first time GetAndRefreshBodyData is called,
               Kinect will allocate each Body in the array.
            // As long as those body objects are not disposed and
                not set to null in the array,
            // those body objects will be re-used.
            bodyFrame.GetAndRefreshBodyData(this.bodies);
            dataReceived = true;
            floorPlane = bodyFrame.FloorClipPlane;
            //Calculation of Elapsed Time
            elapsedTime = bodyFrame.RelativeTime.TotalSeconds -
               lastFrameTime.TotalSeconds;
            lastFrameTime = bodyFrame.RelativeTime;
        }
    }
    if (dataReceived)
    {
        using (DrawingContext dc = this.drawingGroup.Open())
        {
            // Draw a transparent background to set the render
               size
            dc.DrawRectangle(Brushes.Black, null, new Rect(0.0,
               0.0, this.displayWidth, this.displayHeight));
            int penIndex = 0;
            //The Closest body is selected
            Body body = bodies.Closest();
            if (body != null)
```

```
{
        Pen drawPen = this.bodyColors[penIndex++];
        if (body.IsTracked)
        {
          this.DrawClippedEdges(body, dc);

          if (refBody == null)
          {
                refBody = new ReferenceBody();
          }
          if (mainBody == null)
          {
                mainBody = new BodyEx();
          }
          if(mainBody != null)
          {
                mainBody.UpdateBody(body);
          }
          //Lean helper for Reference Setting
          string bodyText = "Body is " + (body.IsStable
             () ? "Stable" : "Not Stable");
          if (body.LeanTrackingState == TrackingState.
             Tracked)
          {
                Point pt = new Point(10, 5);
                bodyText = String.Format("Body lean
                   {0:0.0000}, {1:0.0000}, Height:
                   {2:0.00}", body.Lean.X, body.Lean.Y,
                    body.Height());
                dc.DrawText(bodyText, pt,
                   defaultFontColor);
```

```
}
if (!refBody.IsReferenceSet)
{
        //Reference set
        refBody.SetReference(body);
        dc.DrawText(String.Format("Ref Count:
            {0}", refBody.ReferenceSetCount),
            new Point(10, 20), defaultFontColor)
            ;
        refBody.FloorPlane = floorPlane;
        mainBody.DrawBody(dc, drawPen,
            trackedJointBrush);
}
else
{
//Exp Smoothing
foreach (var item in mainBody.Joints)
{
        JointInfo ji = item.Value;
        expSmooth.Update(ref ji.
            CameraSpacePosition, ji.
            TrackingState);
}
mainBody.DrawBody(dc, drawPen,
    trackedJointBrush);
walkingBody.Update(mainBody, elapsedTime);
JointType[] jts = { JointType.FootLeft,
    JointType.FootRight, JointType.SpineBase
    };
foreach (var jt in jts)
{
```

```
                JointInfo jii = mainBody[jt];
                var euo = jii.Orientation.Orientation.
                    ToEuler();
                double []euler= { euo.X, euo.Y, euo.Z};
                euler[0] = euler[0].ToDegrees();
                euler[1] = euler[1].ToDegrees();
                euler[2] = euler[2].ToDegrees();
            }
            var eulerOri = mainBody[JointType.SpineBase].
                Orientation.Orientation.ToEuler();
            if ((walkingBody.RightFootMoving ||
                walkingBody.LeftFootMoving) && walkingBody
                .SpineMoving)
            {
                var eulerYAngles = ((double)eulerOri.Y)
                    .ToDegrees();
                txtInfo.Text = String.Format("Angle:
                    {0:0.00}, Speed: {1:0.00 m/s}",
                eulerYAngles, walkingBody.
                    SpineBaseTracker.GaussianSpeed());
            }
            else
            {
                txtInfo.Text = "";
            }
            }

        }
    }

    // prevent drawing outside of our render area
```

```
            this.drawingGroup.ClipGeometry = new
                RectangleGeometry(new Rect(0.0, 0.0, this.
                displayWidth, this.displayHeight));
        }
    }
}
```

Code for PositionTracker

```
public class PositionTracker
{
private :
  Queue<Tuple<Vector3,double>> positionSpeedQueue = new Queue<
      Tuple<Vector3, double>>();
  Vector3 lastPosition;
  double lastSpeed = 0;
  double lastGaussianSpeed = 0;
  double lastExponentialSpeed = 0;

  //Calculation of Simple Speed based on geometric distance
  double SimpleSpeed(Vector3 firstPosition, Vector3
      lastPosition, double elapsedTime)
  {
    float distance = Vector3.Distance(firstPosition,
        lastPosition);
    return distance / elapsedTime;
  }

  //Calculation of Gaussian Speed
  double GaussianSpeed(Queue<Tuple<Vector3, double>> posSpeed)
  {
```

```
    double speed = 0;
    if(posSpeed.Count >= SelectedKernel.Length)
    {
      var arr = posSpeed.ToArray();
      //Multiplying selected kernel with the speed values
      for (int i = 0; i < posSpeed.Count; i++)
      {
        speed += SelectedKernel[i] * arr[i].Item2;
      }
    }
    return speed;
  }
  double ProcessJointPosition(SharpDX.Vector3 position, double
      elapsedTime)
  {
    if(positionSpeedQueue.Count >= SelectedKernel.Length)
    {
      var firstPos = positionSpeedQueue.Dequeue();
    }
    //Calculating Simple speed unless otherwise stated
    double curSpeed = SimpleSpeed(lastPosition, position,
        elapsedTime);
    //Enqueue values
    positionSpeedQueue.Enqueue(new Tuple<Vector3, double>(
        position, curSpeed));
    //Save this position and speed as last position to easiliy
        calculate the speed and acceleration
    this.lastPosition = position;
    lastSpeed = curSpeed;
    lastGaussianSpeed = 0;
    lastExponentialSpeed = 0;
```

```
    return curSpeed;
  }
}
```

Code for ExponentialJoint

Largely based on the work of [45]

```
public class ExponentialSmooth
{
  float msmoothing,
    mcorrection,
    mprediction,
    mjitterRadius,
    mmaxDeviationRadius;
  FilterDoubleExponentialData mhistory;

  void Update(ref CameraSpacePoint point, TrackingState ts)
  {
    // If not tracked, we smooth a bit more by using a bigger
       jitter radius
    // Always filter feet highly as they are so noisy
    var jitterRadius = mjitterRadius;
    var maxDeviationRadius = mmaxDeviationRadius;
    if (ts != TrackingState.Tracked)
    {
      jitterRadius *= 2.0f;
      maxDeviationRadius *= 2.0f;
    }
    Vector3 filteredPosition,
            positionDelta,
            trend;
```

```
float diffVal;
var reportedPosition = new Vector3(point.X, point.Y, point.
    Z);
var prevFilteredPosition = mhistory.FilteredPosition;
var prevTrend = mhistory.Trend;
var prevRawPosition = mhistory.ReportedPosition;
// If joint is invalid, reset the filter
if (reportedPosition.X == 0 && reportedPosition.Y == 0 &&
    reportedPosition.Z == 0)
{
  Reset();
}
// Initial start values
if (this.mhistory.FrameCount == 0)
{
  filteredPosition = reportedPosition;
  trend = new Vector3();
}
else if (this.mhistory.FrameCount == 1)
{
  filteredPosition = (reportedPosition + prevRawPosition) *
      0.5f;
  positionDelta = filteredPosition - prevFilteredPosition;
  trend = (positionDelta * mcorrection) + (prevTrend * (1.0
     f - mcorrection));
}
else
{
  // First apply jitter filter
  positionDelta = reportedPosition - prevFilteredPosition;
  diffVal = Math.Abs(positionDelta.Length());
```

```
    if (diffVal <= jitterRadius)
    {
      filteredPosition = (reportedPosition * (diffVal /
          jitterRadius)) +
      (prevFilteredPosition * (1.0f - (diffVal / jitterRadius
          )));
    }
    else
    {
      filteredPosition = reportedPosition;
    }
    // Now the double exponential smoothing filter
    filteredPosition = (filteredPosition * (1.0f - msmoothing
        )) +
    ((prevFilteredPosition + prevTrend) * msmoothing);

    positionDelta = filteredPosition - prevFilteredPosition;
    trend = (positionDelta * mcorrection) + (prevTrend * (1.0
        f - mcorrection));
  }
  // Predict into the future to reduce latency
  var predictedPosition = filteredPosition + (trend *
      mprediction);
  // Check that we are not too far away from raw data
  positionDelta = predictedPosition - reportedPosition;
  diffVal = Math.Abs(positionDelta.Length());
  if (diffVal > maxDeviationRadius)
  {
    predictedPosition = (predictedPosition * (
        maxDeviationRadius / diffVal)) +
```

```
            (reportedPosition * (1.0f - (maxDeviationRadius / diffVal
                )));
        }
        // Save the data from this frame
        this.mhistory.ReportedPosition = reportedPosition;
        this.mhistory.FilteredPosition = filteredPosition;
        this.mhistory.Trend = trend;
        this.mhistory.FrameCount++;

        // Set the filtered data back into the joint
        point.X = predictedPosition.X;
        point.Y = predictedPosition.Y;
        point.Z = predictedPosition.Z;
    }

    private struct FilterDoubleExponentialData
    {
    public Vector3 ReportedPosition;
    public Vector3 FilteredPosition;
    public Vector3 Trend;
    public uint FrameCount;
    }
}
```

Matlab Code for analysis

```
function test1_orientation_positionFn( input_args )

clear all;close all;main_data=load('test1_orientation_position.
    txt');
```

```
spine = main_data(3:3:end,:);
spinePosition = spine(:,4:6);
spineOrientation = spine(:,1:3);
data_time = spine(:,7); % In second
% Calculate 3D distance
linewidth = 1.2;
data3d = spinePosition;
data_time = spine(:,7); % In second
%data_time(1) = 1;
%t2 = 1:size(data_time,1);
t2 = cumsum(data_time);

gwindowSize = 7;
fswindowSize = 7;
maKern = ones(1, gwindowSize)/gwindowSize;
spd3d = speed3D(data3d, data_time);

spd3d_fs = fastsmooth(spd3d, fswindowSize, 2, 1);
spd3d_ma = filter(maKern, 1, spd3d);
spd3d_g = doGauss(spd3d, 7);

corr_g = corr(spd3d_g, spd3d);
corr_fs = corr(spd3d_fs, spd3d);
corr_ma = corr(spd3d_ma, spd3d);
[corr_g corr_fs corr_ma ]

mse_g = sum((spd3d_g-spd3d).^2)/size(spd3d,1);
mse_fs = sum((spd3d_fs-spd3d).^2)/size(spd3d,1);
mse_ma = sum((spd3d_ma-spd3d).^2)/size(spd3d,1);
[mse_g mse_fs mse_ma]
```

```
figure, plot(t2, spd3d, t2, spd3d_g, 'linewidth', linewidth),
title('Spine Speed using Gaussian (window 7)'),
xlabel('Time (in seconds)'),ylabel('Speed (in ms^{-1})'),
legend('Original Speed','Smoothed Speed')

figure, plot(t2, spd3d, t2, spd3d_ma, 'linewidth', linewidth),
title('Spine Speed using Moving Average (window 7)')
xlabel('Time (in seconds)'),ylabel('Speed (in ms^{-1})')
legend('Original Speed','Smoothed Speed')

figure, plot(t2, spd3d, t2, spd3d_fs, 'linewidth', linewidth),
title('Spine Speed using Triangular (window 7)')
xlabel('Time (in seconds)'),ylabel('Speed (in ms^{-1})')
legend('Original Speed','Smoothed Speed')

%accel
accel = zeros(size(spd3d));
accel_g = zeros(size(spd3d));
accel_fs = zeros(size(spd3d));
accel_ma = zeros(size(spd3d));

accel(2:end,:) = diff(spd3d)./data_time(2:end);
accel_fs(2:end,:) = diff(spd3d_fs)./data_time(2:end);
accel_fs = fastsmooth(accel_fs, fswindowSize+2);
accel_g(2:end,:) = diff(spd3d_g)./data_time(2:end);
accel_g = doGauss(accel_g, gwindowSize+2);
accel_ma(2:end,:) = diff(spd3d_ma)./data_time(2:end);
accel_ma = filter(maKern, 1, accel_ma);

corr_ag = corr(accel_g, accel);
corr_afs = corr(accel_fs, accel);
```

```
corr_ama = corr(accel_ma, accel);
display('------')
[corr_ag corr_afs corr_ama ]

mse_ag = sum((accel_g-accel).^2)/size(accel,1);
mse_afs = sum((accel_fs-accel).^2)/size(accel,1);
mse_ama = sum((accel_ma-accel).^2)/size(accel,1);
[mse_ag mse_afs mse_ama]

ori_y = spineOrientation(:,2);
ori_y_fs = fastsmooth(ori_y, fswindowSize, 2, 1);
ori_y_g = doGauss(ori_y, gwindowSize);
figure, plot(t2, accel_g, t2, ori_y, 'linewidth', linewidth),
    title('tt');
figure, plot(t2, accel_g, t2, ori_y_fs, 'linewidth', linewidth)
    , title('tt2');
figure, plot(t2, accel_g, 'linewidth', linewidth)
title('Spine Acceleration Smoothed with Gaussian')
xlabel('Time (in seconds)'),ylabel('Acceleration (in ms^{-2})')

figure, plot(t2, ori_y_g, 'linewidth', linewidth);
title('Spine Cumulative Angle smoothed with Gaussian')
xlabel('Time (in seconds)'),ylabel('Angle (in Degree^{\circ})')



figure, plot(t2, accel, t2, accel_fs, 'linewidth', linewidth),
title('Spine Acceleration using Triangular smoothing (window 7)
    ')
xlabel('Time (in seconds)'),ylabel('Acceleration (in ms^{-2})')
legend('Original Acceleration','Smoothed Acceleration');
```

```
figure, plot(t2, accel, t2, accel_g, 'linewidth', linewidth),
    title('Accel vs accel_g');
title('Spine Acceleration using Gaussian smoothing (window 7)')
xlabel('Time (in seconds)'),ylabel('Acceleration (in ms^{-2})')
legend('Original Acceleration','Smoothed Acceleration');

figure, plot(t2, accel, t2, accel_ma, 'linewidth', linewidth),
    title('Accel vs accel_ma');
title('Spine Acceleration using Moving Average smoothing (
    window 7)')
xlabel('Time (in seconds)'),ylabel('Acceleration (in ms^{-2})')
legend('Original Acceleration','Smoothed Acceleration');



figure, plot(t2, accel_fs, t2, accel_ma, 'linewidth', linewidth
    ), title('Accel_fs vs accel_ma');
figure, plot(t2, accel_g, t2, accel_ma, 'linewidth', linewidth)
    , title('Accel_g vs accel_ma');
figure, plot(t2, accel_g, t2, accel_fs, 'linewidth', linewidth)
    , title('Accel_g vs accel_fs');
end
```

REFERENCES

[1] G. K. Edgar, "Accommodation, cognition, and virtual image displays: A review of the literature," *Displays*, vol. 28, no. 2, pp. 45–59, 2007.

[2] M. Fritschi, H. Esen, M. Buss, and M. O. Ernst, "Multi-modal vr systems," in *The Sense of Touch and Its Rendering*. Springer, 2008, pp. 179–206.

[3] E. Richard, A. Tijou, P. Richard, and J.-L. Ferrier, "Multi-modal virtual environments for education with haptic and olfactory feedback," *Virtual Reality*, vol. 10, no. 3-4, pp. 207–225, 2006.

[4] A. Kemeny, "Driving simulation for virtual testing and perception studies," in *Proceedings of DSC Europe Conference, Monte-Carlo*, 2009, pp. 15–23.

[5] H. Teufel, H.-G. Nusseck, K. Beykirch, J. Butler, M. Kerger, and H. Bülthoff, "Mpi motion simulator: development and analysis of a novel motion simulator," in *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 2007, p. 6476.

[6] N. E. Seymour, "Vr to or: a review of the evidence that virtual reality simulation improves operating room performance," *World journal of surgery*, vol. 32, no. 2, pp. 182–188, 2008.

[7] C. V. Erren-Wolters, H. van Dijk, A. C. de Kort, M. J. IJzerman, and M. J. Jannink, "Virtual reality for mobility devices: training applications and clinical results: a review," *International Journal of Rehabilitation Research*, vol. 30, no. 2, pp. 91–96, 2007.

[8] F. P. Vidal, F. Bello, K. W. Brodlie, N. W. John, D. Gould, R. Phillips, and N. J. Avis, "Principles and applications of computer graphics in medicine," in *Computer Graphics Forum*, vol. 25, no. 1. Wiley Online Library, 2006, pp. 113–137.

[9] M. V. Sanchez-Vives and M. Slater, "From presence to consciousness through virtual reality," *Nature Reviews Neuroscience*, vol. 6, no. 4, pp. 332–339, 2005.

[10] R. A. Ruddle and S. Lessels, "For efficient navigational search, humans require full physical movement, but not a rich visual scene," *Psychological Science*, vol. 17, no. 6, pp. 460–465, 2006.

[11] R. Bertin, C. Collet, S. Espié, and W. Graf, "Objective measurement of simulator sickness and the role of visual-vestibular conflict situations," in *Driving Simulation Conference North America*, 2005, pp. 280–293.

[12] N. B. Epstein, "Omnidirectional moving surface," Jun. 1 2004, uS Patent 6,743,154.

[13] J. L. Souman, P. R. Giordano, M. Schwaiger, I. Frissen, T. Thümmel, H. Ulbrich, A. D. Luca, H. H. Bülthoff, and M. O. Ernst, "Cyberwalk: Enabling unconstrained omnidirectional walking through virtual environments," *ACM Transactions on Applied Perception (TAP)*, vol. 8, no. 4, p. 25, 2011.

[14] A. De Luca, R. Mattone, P. R. Giordano, and H. H. Bülthoff, "Control design and experimental evaluation of the 2d cyberwalk platform," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on.* IEEE, 2009, pp. 5051–5058.

[15] K. J. Fernandes, V. Raja, and J. Eyre, "Cybersphere: the fully immersive spherical projection system," *Communications of the ACM*, vol. 46, no. 9, pp. 141–146, 2003.

[16] J.-Y. Huang, "An omnidirectional stroll-based virtual reality interface and its application on overhead crane training," *IEEE Transactions on Multimedia*, vol. 5, no. 1, pp. 39–51, 2003.

[17] KatVR. (2015, jul) Katwalk VR premium. [Online]. Available: http://www.katvr.com/gailan.html

[18] VirtuiX. (2016, jan) Virtuix omni VR platform. [Online]. Available: http://www.virtuix.com/

[19] M. Hoffmann, K. Schuster, D. Schilberg, and S. Jeschke, "Next-generation teaching and learning using the virtual theatre," in *Automation, Communication and Cybernetics in Science and Engineering 2015/2016.* Springer, 2016, pp. 281–291.

[20] M. Schwaiger, T. Thuimmel, and H. Ulbrich, "Cyberwalk: An advanced prototype of a belt array platform," in *Haptic, Audio and Visual Environments and Games, 2007. HAVE 2007. IEEE International Workshop on.* IEEE, 2007, pp. 50–55.

[21] N. Silberman and R. Fergus, "Indoor scene segmentation using a structured light sensor," in *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on.* IEEE, 2011, pp. 601–608.

[22] E. Lachat, H. Macher, T. Landes, and P. Grussenmeyer, "Assessment and calibration of a rgb-d camera (kinect v2 sensor) towards a potential use for close-range 3d modeling," *Remote Sensing*, vol. 7, no. 10, pp. 13 070–13 097, 2015.

[23] H. Sarbolandi, D. Lefloch, and A. Kolb, "Kinect range sensing: Structured-light versus time-of-flight kinect," *Computer Vision and Image Understanding*, vol. 139, pp. 1–20, 2015.

[24] L. Yang, L. Zhang, H. Dong, A. Alelaiwi, and A. El Saddik, "Evaluating and improving the depth accuracy of kinect for windows v2," *IEEE Sensors Journal*, vol. 15, no. 8, pp. 4275–4285, 2015.

[25] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison *et al.*, "Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera," in *Proceedings of the 24th annual ACM symposium on User interface software and technology.* ACM, 2011, pp. 559–568.

[26] J. A. Diego-Mas and J. Alcaide-Marzal, "Using kinect sensor in observational methods for assessing postures at work," *Applied ergonomics*, vol. 45, no. 4, pp. 976–985, 2014.

[27] S. Asteriadis, A. Chatzitofis, D. Zarpalas, D. S. Alexiadis, and P. Daras, "Estimating human motion from multiple kinect sensors," in *Proceedings of the 6th international conference on computer vision/computer graphics collaboration techniques and applications*. ACM, 2013, p. 3.

[28] X. Xu, R. W. McGorry, L.-S. Chou, J.-h. Lin, and C.-c. Chang, "Accuracy of the microsoft kinect for measuring gait parameters during treadmill walking," *Gait & posture*, vol. 42, no. 2, pp. 145–151, 2015.

[29] Microsoft. (2014, mar) Microsoft Kinect SDK. [Online]. Available: https://developer.microsoft.com/en-us/windows/kinect/

[30] J. Sell and P. O'Connor, "The xbox one system on a chip and kinect sensor," *IEEE Micro*, vol. 34, no. 2, pp. 44–53, 2014.

[31] C. S. Bamji, P. O'Connor, T. Elkhatib, S. Mehta, B. Thompson, L. A. Prather, D. Snow, O. C. Akkaya, A. Daniel, A. D. Payne *et al.*, "A 0.13 μm cmos system-on-chip for a 512$\times$ 424 time-of-flight image sensor with multi-frequency photo-demodulation up to 130 mhz and 2 gs/s adc," *IEEE Journal of Solid-State Circuits*, vol. 50, no. 1, pp. 303–319, 2015.

[32] L. Valgma, "3d reconstruction using kinect v2 camera," Ph.D. dissertation, Tartu Ülikool, 2016.

[33] P. Sturm, "Pinhole camera model," in *Computer Vision*. Springer, 2014, pp. 610–613.

[34] B. Galna, G. Barry, D. Jackson, D. Mhiripiri, P. Olivier, and L. Rochester, "Accuracy of the microsoft kinect sensor for measuring movement in people with parkinson's disease," *Gait & posture*, vol. 39, no. 4, pp. 1062–1068, 2014.

[35] C.-Y. Chang, B. Lange, M. Zhang, S. Koenig, P. Requejo, N. Somboon, A. A. Sawchuk, and A. A. Rizzo, "Towards pervasive physical rehabilitation using microsoft kinect," in *Pervasive Computing Technologies for Healthcare (PervasiveHealth), 2012 6th International Conference on*. IEEE, 2012, pp. 159–162.

[36] M. Elgendi, F. Picon, N. Magnenat-Thalmann, and D. Abbott, "Arm movement speed assessment via a kinect camera: a preliminary study in healthy subjects," *Biomedical engineering online*, vol. 13, no. 1, p. 88, 2014.

[37] M. Elgendi, F. Picon, and N. Magenant-Thalmann, "Real-time speed detection of hand gesture using, kinect," in *Proc. Workshop on Autonomous Social Robots and Virtual Humans, The 25th Annual Conference on Computer Animation and Social Agents (CASA 2012)*, 2012.

[38] J. Shotton, T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore, "Real-time human pose recognition in parts from single depth images," *Communications of the ACM*, vol. 56, no. 1, pp. 116–124, 2013.

[39] J. Han, L. Shao, D. Xu, and J. Shotton, "Enhanced computer vision with microsoft kinect sensor: A review," *IEEE transactions on cybernetics*, vol. 43, no. 5, pp. 1318–1334, 2013.

[40] M. Gabel, R. Gilad-Bachrach, E. Renshaw, and A. Schuster, "Full body gait analysis with kinect," in *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*. IEEE, 2012, pp. 1964–1967.

[41] Midori Kitagawa. (2010) Dynamic walk representations in graphs. [Online]. Available: http://www.utdallas.edu/atec/midori/Handouts/walkingGraphs.htm#GC

[42] Infinadeck. (2016, jan) Infinadeck at CES 2016. [Online]. Available: https://www.facebook.com/Infinadeck/

[43] Microsoft. (2015, jan) Xbox playspace setup. [Online]. Available: https://support.xbox.com/en-US/xbox-360/accessories/kinect-sensor-setup#e654fb0055954e9787040698b82b3591

[44] K. R. Kaufman and D. H. Sutherland, "Kinematics of normal human walking," *Human walking*, vol. 3, pp. 33–52, 2006.

[45] Paul York. (2015) Kinect extended library for kinect. [Online]. Available: https://github.com/KinectEx/KinectEx