DEVELOPMENT AND EVALUATION OF AN ENHANCED WEIGHTED FREQUENCY FOURIER LINEAR COMBINER ALGORITHM USING BANDWIDTH INFORMATION

by

Wonchul Nho

BS, Drexel University, 1993

MS, University of Pittsburgh, 1995

Submitted to the Graduate Faculty of

School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2006

UNIVERSITY OF PITTSBURGH

SCHOOL OF ENGINEERING

This dissertation was presented

by

Wonchul Nho

It was defended on

December 16, 2005

and approved by

Ching-Chung Li, Professor, Electrical and Computer Engineering

Marlin H. Mickle, Professor, Electrical and Computer Engineering

Heung-no Lee, Assistant Professor, Electrical and Computer Engineering

Rory A. Cooper, Professor, Rehabilitation Science and Technology

Dissertation Co-Advisors: J. Robert Boston, Professor, Electrical and Computer Engineering

David M. Brienza, Associate Professor, Rehabilitation Science and Technology

DEVELOPMENT AND EVALUATION OF AN ENHANCED WEIGHTED FREQUENCY FOURIER LINEAR COMBINER ALGORITHM USING BANDWIDTH INFORMATION IN JOYSTICK OPERATION

Wonchul Nho, PhD

University of Pittsburgh, 2006

Driving an electric powered wheelchair with a joystick is a complex task for the user who has a pathological tremor. Most powered wheelchairs use simple filtering algorithms to reduce the effects of tremor. These algorithms work well in most situations, but fall short in others. This study addresses the problems associated with pathological tremor associated with Cerebral Palsy (CP). The purpose of this study is to know more about the characteristics of CP tremor with time-frequency analysis and to improve joystick control with other advanced filtering algorithms.

We used three estimated parameters, instantaneous frequency (IF), instantaneous bandwidth (IB), and instantaneous power (IP), from a time-frequency distribution (TFD), to characterize CP tremor and to tune a notch filter for canceling CP tremor noise from a joystick signal in an off-line experiment. From the off-line experiment, we showed that our CP tremor suppression system performed better with the information of IF, IB, and IP.

We also conducted an on-line experiment in which we introduced two tremor suppression algorithms. One is Weighted-frequency Fourier Linear Combiner (WFLC), which estimates a tremor frequency and its weight, and the other is our modified WFLC, which adjusts a notch width with respect to the bandwidth of CP tremor additionally. We implemented both algorithms on the virtual wheelchair driving test along with a commonly used low-pass filter. We recruited ten subjects who have CP tremor and tested them in a virtual wheelchair driving environment.

We observed that CP tremors in the joystick signal were suppressed greatly by the new filtering algorithms. We learned that the time-delay of a low-pass filter caused serious performance degradation of wheelchair driving and observed that most subjects performed better with new filtering methods than with a low-pass filter. Furthermore, since our modified WFLC algorithm was able to reduce more CP tremor noise than WFLC, we learned that it is important to consider the bandwidth information of CP tremor when designing a tremor suppression system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

I would like to thank my advisors, Drs. David M. Brienza and J. Robert. Boston, for their guidance during my PhD study. I also thanks to Drs. Ching-Chung Lee, Marlin H. Mickle, Heung-no Lee, and Rory A. Cooper for taking their time to serve on my thesis committee and for their suggestions.

I would also like to thank HERL, especially Drs. Cooper, Spaeth and Ding, allowing us to use his isometric joystick and a virtual driving software package. Special thanks go to Ms. Jeanne Zanka and Dr. Wonkoo Kim, my peer reviewers for my documents.

I particularly owe my success to my wife, Mila, and my three daughters, Hannah, Bona, and Laena, for being extremely patient and supportive for the past years. I would like to thanks to my parents, Heesin and Jaeso, for their endless support.

1.0 INTRODUCTION

1.1 RATIONALE OF PATHOLOGICAL TREMOR STUDY

Many people with disabilities use electric powered wheelchairs for their mobility. Independent mobility reduces dependence on caregivers and family members and promotes feelings of self-reliance [1]. However, driving an electric powered wheelchair with a joystick is a complex task for the user who has a pathological tremor (including athetoid movement). Pathological tremor is the most common of all motor disorders. Over three million people in North America are affected by pathological tremor [1]. Pathological tremor can prevent the user from generating a reliable continuous input signal to the joystick controller. In the worst case, tremor places the wheelchair driver at risk for injury, at best, tremor reduces driving efficiency. Most power wheelchairs use simple filtering algorithms to reduce the effects of tremor. These algorithms work well in most situations, but fall short in others. This study will address the problems associated with a pathological tremor associated with Cerebral Palsy (CP). CP is defined as a group of disorders characterized by loss of movement or loss of other nerve functions. CP results from injury to the cerebrum that occurs during fetal development or near the time of birth.

To develop a new filtering algorithm for tremor, it is necessary to first know the precise characteristics of tremor. Frequency and amplitude are major elements used to characterize tremor. Many researchers, [1], [2], and [3], have used power spectral density to characterize the frequency and peak amplitude of tremor and have used this frequency information to filter out unwanted tremor noise from the desired signal. However, precise characterization of the frequencies and amplitudes of pathologic tremor are not well defined because the frequency varies depending on patient, posture and patient's condition [2].

Since the tremor signal possesses spectral characteristics that vary with time, the spectral density of the signal indicates only that frequencies occurred, but does not indicate when they occurred. Therefore, the power spectral density alone does not enable one to see the whole behavior of the time-varying signal. We investigated this problem and other characteristics of the tremor by using a joint time-frequency description of the signals. An alterative technique based on time-frequency analysis may be a better solution. Time-frequency analysis describes a relatively new signal processing technique where instantaneous frequency (IF), instantaneous bandwidth (IB), and instantaneous power (IP) of a signal are computed. The use of such parameters is likely to give a more complete and accurate characterization of pathological tremor. Just as the spectral and temporal densities, respectively, indicate the intensity of the signal as a function of frequency and the intensity of the signal as a function of time, a time-frequency density ideally indicates the intensity of the signal as a function of time and frequency [6]. With time-frequency analysis, we are able to find when and where the pathological tremor occurs.

Although many researchers have only concentrated on finding frequency information of the pathological tremor for their filtering algorithms, we consider bandwidth information as an additional constraint of the filter design since tremor amplitude has a time-varying envelope and the bandwidth of tremor varies with respect to the envelope [5]. We need to design a filter that responds to time-varying bandwidth information as well as time-varying frequency information of CP tremor. We investigated the possibility of using IF, IB, and IP as candidates of the filter constraints to improve the suppression of pathological tremor noise in the joystick signal of electric powered wheelchair. Since time-frequency distribution (TFD) is typically used as a time-varying spectral estimation method, we used TFD to characterize the pathological tremor by estimating the IF, IB, and IP of the joystick tremor noise.

The most popular filtering algorithm in the joystick of power wheelchairs is a low-pass filter with 3 Hz cut-off frequency. A serious drawback of using low-pass filter is the problem of time-delay. In addition, intentional joystick movement is often at a lower frequency than pathological tremor, but intentional joystick movement ranges as high as 5 Hz or as low as 2 Hz for CP tremor would also be distorted by a typical low-pass filter. In this case, a notch filter may be more suitable. A notch filter only suppresses the tremor frequency rather than all frequencies beyond a certain point as is done by the low-pass filter. The notch filter would, therefore, cause less distortion of the intentional movement signal. However, frequency and amplitude of tremor are unknown before hand and are time-varying, so a fixed notch filter is generally inappropriate. Therefore, it is beneficial to implement a time-varying notch filter where the algorithm learns the frequency and amplitude of the tremor in order to suppress it.

In this study, we introduced two filtering techniques: (1) a time-varying notch filter with TFD technique, and (2) an adaptive notch filter based on Fourier Linear Combiner (FLC) technique. The time-varying notch filter method uses information gained from the TFD, whereby a filter notch is tuned to the IF, IB, and IP of the tremor noise. The time-varying notch filter with TFD estimates a tremor frequency, tremor bandwidth, and tremor amplitude in order to suppress unwanted tremor noise. The TFD method works very well in off-line implementation, but it is not ideal in on-line implementation because of problems with time-delay and computational complexity.

In contrast to the time-varying notch filter described above, the adaptive notch algorithm is well suited to active tremor control due to its computational simplicity and predictive capability. Riviere's Weighted-frequency Fourier Linear Combiner (WFLC) and our modified WFLC were implemented as an adaptive notch filter in a real-time tremor suppression system and we compared their results to that of the low-pass filter. Riviere's WFLC acts like an adaptive notch filter that controls notch frequency and notch depth, but our modified WFLC controls one additional element: notch bandwidth. We implemented both algorithms on a virtual wheelchair driving test along with a commonly used low-pass filter and compared their performances. We recruited ten subjects who have CP tremor and tested them in a virtual wheelchair driving environment.

1.2 OUTLINE

This dissertation is organized as follows. Relevant literature and background of time-frequency analysis for this study are summarized in Chapter 2. Current characteristics of pathological tremor with CP were revealed and current tremor suppression methods were reviewed. Chapter 3 describes a time-varying notch filter for the off-line experiment. Several IF estimation methods including IF estimation based on the TFD technique were reviewed. The importance of IB in the time-varying notch filter was demonstrated with a synthetic signal and preliminary joystick data. Chapter 4 describes an adaptive notch filter technique for the on-line experiment. We introduced Riviere's WFLC and our modified WFLC as a tremor suppression method. Chapter 5 describes the test settings, including human subjects and experimental hardware and software setups.

Results for the virtual driving experiment are presented in Chapter 6. Finally, implications of this study and future research areas are discussed in Chapter 7.

2.0 BACKGROUND

In this chapter, we review the previous work on tremor characterization and classification, tremor suppression schemes for electric powered wheelchair (EPW) joystick use, and other tremor suppression algorithms. In addition, we review the fundamentals of time-frequency distribution with the definition of instantaneous frequency (IF), instantaneous bandwidth (IB), and instantaneous power (IP).

2.1 CHARACTERISTICS OF TREMOR

Tremor is an involuntary, approximately rhythmic, and roughly sinusoidal movement [2], [7], [8], [10]. There are two types of common tremor, physiological and pathological. Normal tremor, commonly known as physiologic tremor, shows very small amplitudes. Its energy is concentrated at frequencies higher than 8-12 Hz [10]. Physiological tremor does not represent a serious problem for most activities of daily living. Pathological tremor, on the other hand, can severely affect activities of daily living. Pathological tremor arises in cases of injury and disease. The power spectrum of pathological tremor is concentrated in frequencies ranging from 2-11 Hz, which is lower in frequency than physiological tremor [1], [10]. In addition, pathological tremor oscillation amplitudes are significantly stronger than those of physiological tremor, introducing uncomfortable distortion into purposeful movements [11]. In many instances, the tremor amplitude can be so severe as to make purposeful movement virtually impossible. Three of the most common causes of pathological tremor are Cerebral Palsy, essential tremor and Parkinson’s disease [7].

The first attempts to characterize tremor are found in the scientific literature of the 1960s. Stiles et al. analyzed hand tremor characteristics using EMG sensors [29]. Since then, more advanced techniques have been developed to analyze tremor. Edwards and Beuter developed indices for identification of abnormal tremor using computer tremor evaluation systems [13]. Eberhart and Hu analyzed hand tremor using particle swarm optimization to distinguish between normal subjects and those with tremor. Then they tried to differentiate between normal physiological tremor and pathological tremors. Despite years of effort, the precise characterization of pathological tremor has not been well defined.

Recently, Gonzalez et al. used frequency range to characterize tremors. Cerebral Palsy tremor is in the 2-6 Hz range, essential tremor is in the 6-11 Hz range, and Parkinson’s tremor is in the 3-7 Hz range typically, while most intended movement is in the 0-1 Hz range [7] (Gillard reported a different range of 0.5-2.5 Hz [1]). In addition, the mechanical relationship between tremor intensity and oscillation frequency has been outlined. According to Aylor et al., the oscillation amplitude diminishes as the mean tremor frequency increases. For the same reason, large amplitude tremors are related to low frequencies [12]. These characteristics were then used to filter the unwanted tremor from the desired input signal [7]. This technique is likely to yield unsatisfactory results due to the time-varying nature of tremor frequency and amplitude [15].

2.2 TREMOR SUPPRESSION SYSTEM FOR ELECTRIC POWERED WHEELCHAIRS

Until recently, only three methods of treating tremor existed: medication, inertial loading, and viscous loading [7]. None of these treatments are ideal. Approximately 50% of people with pathological tremor do not respond to medication. Viscous and inertial loads provide resistance to motion which increases with frequency for viscous damping and the square of frequency for inertial loading. Inertial loads are often used to selectively reduce tremor but they also attenuate functional movements to some extent and cause muscle fatigue [1], [8]. As an extension of an inertial loading study, some researchers [4], [9] developed a force-sensing joystick as an input device to the electric powered wheelchair because they believed that an isometrically restrained input device limits the effects of tremor. However, Cooper et al. suggested that the isometric joystick may be an acceptable alternative for new wheelchair users, but found no significant differences between the two joysticks [4], [28]. Riley and Rosen also suggested that filtering and control algorithms may be more important than the type of interface [3].

Most current electric powered wheelchairs use simple filtering algorithms and dead zones to reduce the effects of tremor. Such algorithms work well in some situations, but are unreliable in many others. To cater to people with a mild tremor, the controller is installed with a time-delay damping mechanism to slow down the response time. This approach has the disadvantage of slowing the wheelchair down, sometimes unnecessarily, as the person's tremor may vary from time to time. In addition, this delay would affect the ability of the driver to navigate safely around objects and through doorways. The primary method for more severe cases of tremor is to lock the joystick in one direction, the direction first chosen. This device does not let its user move the joystick to the left or right while moving in the forward direction, for example. The obvious disadvantage of this method is the difficulty in guiding the wheelchair smoothly.

Aylor investigated tremor during EPW control through the use of signal processing [12]. His approach was to amplify, rectify and then time average the signal to reduce the effect of tremor on control. The time averaging rate was used to compensate for varying tremor frequencies. Other researchers added a time delay to slow down the response time and then used averaging or a low pass filter (LPF) on the input signal [14]. The problem with this approach is that low frequency intentional inputs are attenuated along with the unintentional tremor input, thus reducing the perceived responsiveness of the wheelchair to the user's inputs. A lack of responsiveness could result in increased instability and could ultimately compromise the user's ability to navigate safely [3], [7], and [11]. Gonzalez and Rahman showed that in some cases, depending on the cutoff frequency and the specific user, low-pass filters can degrade performance [7]. Their final conclusion is that subject-customized filter design is needed.

2.3 ADVANCED TREMOR SUPPRESSION ALGORITHMS FOR OTHER APPLICATIONS AND OTHER TYPES OF TREMOR

A number of digital filtering algorithms have been developed for the purpose of removing unwanted tremor and noise from signals of interest and have found application in tremor suppression. Since in the proposed task we will concentrate on digital filtering techniques for reducing the effects of tremor in an EPW control, a brief review of the relevant literature from other applications is presented as follows.

Some researchers introduced a notch filter to suppress the tremor frequency selectively [11], [16]. However, like LPFs, notch filters are not sufficient due to the time-varying nature of the tremor amplitude and frequency. Reley and Rosen also suggested that filter settings need to be customized for each user [3].

Riviere and Thakor have investigated the application of adaptive notch filtering for the purpose of suppressing pathological tremor noise during computer pen input, and they have developed weighted frequency Fourier linear combiner (WFLC) for computer handwriting [11], [15]. WFLC is an adaptive noise canceller that models tremor. It is used to model the pathological tremor as a modulated sine wave with time-varying frequency. This algorithm estimates tremor frequency as well as amplitude and adapts filtering operations to cancel the tremor. We will revisit this algorithm in Chapter 4 in detail.

2.4 TIME-FREQUENCY DISTRIBUTION

A signal (including tremor signal) is expanded in term of sinusoids of different frequencies

$$s(t) = \frac{1}{\sqrt{2\pi}} \int S(\omega) e^{j\omega t} d\omega . \tag{2-1}$$

The signal is made up of the addition (linear superposition) of the simple wave forms, $e^{j\omega t}$, each characterized by the frequency, ω, and contributing a relative amount indicated by the coefficient, $S(\omega)$. $S(\omega)$ is obtained from the signal by

$$S(\omega) = \frac{1}{\sqrt{2\pi}} \int s(t) e^{-j\omega t} dt \tag{2-2}$$

where $S(\omega)$ is Fourier transform of $s(t)$ and $\left|S(\omega)\right|^2$ is the energy density spectrum.

Pathological tremors possess spectral characteristics that vary with time. The energy density spectrum of the signal tells us which frequencies existed during the total duration of the signal. It gives us no indication as to when these frequencies existed. To acquire this information, a joint time-frequency description of the signals is needed. In this research, we have used the Short-Time Fourier Transform (STFT) (or Spectrogram) as a candidate of TFD, $P(t, \omega)$, because of its simplicity.

Spectrogram is defined by

$$P(t,\omega) = \left|S_t(\omega)\right|^2 = \left|\frac{1}{\sqrt{2\pi}}\int e^{-j\omega\tau}\, s(\tau)h(\tau - t)\,d\tau\right|^2 \qquad (2\text{-}3)$$

where $h(t)$ is the window function. Gaussian window function, $h(t) = \left(\frac{a}{\pi}\right)^{1/4} \exp\left[-\frac{at^2}{2}\right]$, is a good candidate because the Fourier transformation of a Gaussian signal is Gaussian itself, simple in mathematical expression.

Although we have used Spectrogram in our study, there are many other time-frequency distributions that could be used, including Wigner-Ville, Positive distribution, Multi-window Spectrogram, and Jong-Williams distribution. We will test those distributions in further research. The Spectrogram is a widely used method for studying non-stationary signals [17].

The average frequency and bandwidth are

$$\langle\omega\rangle = \int \omega\, P(t,\omega)\, d\omega \text{ and} \qquad (2\text{-}4)$$

$$\sigma_\omega^2 = \int \left(\omega - \langle\omega\rangle\right)^2 P(t,\omega)\, d\omega = \left\langle\omega^2\right\rangle - \langle\omega\rangle^2 . \qquad (2\text{-}5)$$

However, we are interested in local quantities of pathological tremor noise such as IF, IB, and IP in the Spectrogram. Especially, IF and IB are well known components in time-frequency study for analyzing the characteristics of any signal.

IF is defined as the derivative of the phase of a complex representation of the signal by Gabor [19]. Generally, a complex signal can be written in the form,

$$s(t) = A(t)\exp[j\varphi(t)], \tag{2-6}$$

where $s(t)$ is often taken to be the analytic signal (computed via the Hilbert transform of the given real signal), $A(t)$ is the amplitude part of $s(t)$, and $\varphi(t)$ is the phase part of $s(t)$.

IF for any signal, $s(t)$, is

$$IF = \varphi'(t)\ , \tag{2-7}$$

its first moment [24]. TFD first moment provides another means of estimating the IF.

The next parameter, IB, is a concept that has been extensively developed by Cohen and Lee [17], [18], particularly in the context of joint time-frequency distributions, where it is taken to be the standard deviation in frequency at a given time, which is the spread in frequency at a particular time.

IB for any signal, $s(t)$,is

$$IB = \left|\frac{A'(t)}{A(t)}\right| \tag{2-8}$$

where $A'(t)$ is the derivative of $A(t)$.

The last parameter, IP of any signal, is generally $|s(t)|^2$ which is same as a time marginal distribution,

$$|s(t)|^2 = \int P(t, \omega)\, d\omega = P(t). \tag{2-9}$$

However, nature does not break up a signal for us in terms of real and imaginary representations or amplitude and phase. Nature only gives us the left-hand side, real part of $s(t)$, [17]. Therefore, we can not just apply Eqs. 2-7 and 2-8 to calculate IF and IB. In our study, we needed to estimate these quantities. Estimating correct IF is the most critical stage in the suppression system because an incorrectly-estimated IF will force the filter notch to be located in the wrong place. If the notch location is incorrect, even a good measurement of IB will not be helpful in achieving the desired results. In the next chapter, we will talk about IF estimation and suppression techniques for the off-line experiment.

3.0 TECHNICAL DEVELOPMENT: TIME-VARYING NOTCH FILTER WITH TFD (OFF-LINE)

Estimation problems are typically classified into two categories, non-parametric and parametric methods. The difference between the two is the existence and use of a “model” which is used to describe the system. If it is explicitly the parameters of the supposed model that are being estimated then we are dealing with a parameter method. If, however, the quantities being estimated are not dependent on the physical nature or structure of the system then we are dealing with a non-parametric model [26]. There is another distinction commonly made in digital signal processing. It is the notion of a real time or “on-line” algorithm as opposed to an “off-line” one. Strictly speaking, an off-line algorithm would be one that is theoretically impossible to implement in real time, for instance a filter which requires future points. Many times the term is used to refer to a batch type estimation approach as opposed to a recursive one. Also, whether an algorithm can be implemented in real time depends only on a processor’s ability to perform the necessary calculations in the allotted time. For example, the FFT is commonly used in real time applications and is a batch type algorithm.

This chapter addresses the instantaneous frequency (IF) estimation techniques and time-varying notch filters which are used for characterization and suppression of pathological tremors in an off-line situation. We will verify the reason why we choose time-frequency distribution (TFD) for IF estimation and a second order linear time-varying (LTV) IIR notch filter.

3.1 IF ESTIMATION TECHNIQUES

The IF needs to be accurately estimated in order to obtain an intended joystick movement. Many definitions of IF have been proposed which have desirable properties in a given situation. Boashash provides comparisons of the different algorithms with noisy non-stationary signals [24]. From the results given it appears that TFD peak estimation provides a good estimate. We will briefly review currently available IF estimation techniques and provide the reasons why estimation via TFD peak was selected. The coverage includes phase differencing of the analytic signal, counting the zero crossing, adaptive estimation methods based on the least mean square (LMS) algorithm, and estimation via TFD peaks.

3.1.1 Phase difference IF estimation

The definition for the IF of a real continuous-time signal is shown in Eq. 2-7. A discrete finite impulse response (FIR) differentiator is used to implement discrete-time IF estimators based on the definition in Eq. 2-7. The discrete-time IF may then be defined as

$$I\widehat{F}(n) = \varphi(t) * d(n) \tag{3-1}$$

where $d(n)$ is the impulse response of an FIR differentiating filer, and $*$ denotes convolution in time.

Such a filter has practical problems since it exaggerates the effects of high frequency noise. Good approximations to the differentiation operation in discrete-time can be obtained by using a phase differencing operation [24]. The forward finite difference (FFD) is a commonly used phase differencing operation:

$$\widehat{IF}(n) = \varphi(n+1) - \varphi(n) \tag{3-2}$$

The phase difference estimator is unbiased, has zero group delay for linear FM signals, and, corresponds to the first moment in frequency of a number of TFD's.

3.1.2 Zero-crossing IF estimation

Zero-crossing IF estimation is extremely simple computationally because it measures the number of zero-crossings. For a sinusoidal signal, the frequency is given by half the inverse of the interval between zero-crossing. The zero-crossing estimator, assuming the first zero-crossing occurs at time index n, can be expressed as

$$\widehat{IF}_1(n) = \frac{1}{2k} \tag{3-3}$$

where k is the number of sample intervals between zero-crossings.

The linear average of the k consecutive FFD estimators may be rewritten as [24]

$$\widehat{IF}(n) = \frac{1}{2k\pi}\left[\varphi(n+k) - \varphi(n)\right] \tag{3-4}$$

Thus the expression for the zero-crossing estimate is simply a linear average of k adjacent FFD estimates. It is extremely simple computationally. However, if the interval between zero-crossing is not an integer number of samples, then in addition to the linear averaging produced by the estimator, quantization "noise" is introduced.

3.1.3 Adaptive IF estimation

The most popular form of adaptive IF estimation is based on modeling the data via a linear predictive process. The method which may be used for this type of estimation is the least mean square (LMS) algorithm. Griffiths proposed an adaptive IF estimation algorithm based on a linear prediction filter which has its coefficients updated with each new sample [25]. Griffith's method is based conceptually on extracting the peak of a short-time linear prediction based spectral estimate. The resulting algorithm, which is based on gradient descent techniques, is quite simple. However, because the recursive algorithm is inherently an IF tracking process, it is unable to respond to very rapid (or noisy) IF changes. The estimate may therefore exhibit significant noise susceptibility. The main advantage of this algorithm is its computational simplicity.

Details of the algorithm are given below. The vector of data samples at time, n, is denoted by

$$z_n = \left[z(n)\, z(n-1) \cdots z(n-L+1)\right]^T \tag{3-5}$$

where L is the linear prediction filter length, and T represents the transpose operation.

The corresponding vector of liner prediction filter coefficients is

$$a_n = \left[a_1(n)\, a_2(n) \cdots a_L(n)\right] \tag{3-6}$$

As each new data sample is processed, the filter coefficients should ideally be updated so as to minimize the mean square prediction error. The IF estimate is determined from the peak of the linear prediction based spectrum, i.e.,

$$\widehat{IF}(n) = \omega_i \text{ which maximizes } \left| 1 + \sum_{k=1}^{L} a_k(n) \exp[-j\omega_i k] \right|^{-2} \tag{3-7}$$

3.1.4 IF estimation based on TFD

General interpretation of IF of a mono-component signal in the time-frequency domain is the average frequency at each time because the first conditional spectral moment of many TFDs equals the IF [17]. Nho and Loughlin proved that IF is the average frequency at each time with a symmetry condition. For the mono-component tremor case, the equality of $\langle\omega\rangle_t = \varphi'(t) = IF$ is always true [20]. Again, estimation of IF from the TFD is

$$\langle\omega\rangle_t = \int \omega P(\omega | t) d\omega = \frac{1}{P(t)} \int \omega P(t,\omega)\, d\omega \tag{3-8}$$

3.1.5 IB estimation based on TFD

Estimating IB from TFD is a simple process. The IB can be obtained from the time-conditional spectral variance of a TFD of the signal [17].

$$\sigma^2_{\omega|t} = \frac{1}{P(t)} \int \left(\omega - \langle\omega\rangle_t\right)^2 P(t,\omega) d\omega = \left\langle \omega^2 \right\rangle_t - \langle\omega\rangle_t^2 \tag{3-9}$$

where $\left\langle \omega^n \right\rangle_t$ is the n^{th} conditional moment

$$\left\langle \omega^n \right\rangle_t = \int \omega^n\, P(\omega | t)\, d\omega = \frac{1}{P(t)} \int \omega^n\, P(t,\omega)\, d\omega \tag{3-10}$$

where $P(t)$ is the time marginal of the TFD, and $P(\omega \mid t)$ is the time-conditional spectral density.

3.2 TIME-VARYING NOTCH FILTER

A notch filter is a filter that contains one or more deep notches or, ideally, perfect nulls in its frequency response characteristics [22]. Notch filters are useful in many applications where specific frequency components must be eliminated. To create a null in the frequency response of a filter at a frequency ω_0, a pair of complex-conjugate zeros on the unit circle at an angle ω_0 is introduced. That is, $z_{1,2} = e^{\pm j\omega_0}$. Thus the system function for an FIR notch filter is simply

$$H(z) = G\left(1 - 2\cos\omega_0 z^{-1} + z^{-2}\right) \tag{3-11}$$

where G is the filter gain at $\omega = 0$.

The problem with a single pair of FIR notch filters is that the notch has a relatively large bandwidth, which means that other frequency components around the desired null are severely attenuated. To reduce the bandwidth of the null, a longer FIR filter is required. Alternatively, by introducing a pair of complex-conjugate poles at $p_{1,2} = re^{\pm j\omega_0}$ in the system function, we can improve on the frequency response characteristics [22]. The effect of the poles is to introduce a resonance in the vicinity of the null and thus to reduce the bandwidth of the notch. In this research, we have used the IIR filter design method instead of lengthy FIR filter design for the adaptive notch filter since the IIR filter can provide better notch capability than the FIR filter and has advantages of better control of filter notch characteristics with a lower filter order.

The traditional transfer function of the second order linear time-invariant (LTI) IIR filter used here is given by

$$H(z,r) = G\,\frac{1-2\cos\omega_0 z^{-1} + z^{-2}}{1-2r\cos\omega_0 z^{-1} + r^2 z^{-2}} \tag{3-12}$$

where ω_0 is the notch frequency and r represents the notch width control parameter in the range of $0 < r < 1$. The notch width control parameter r is set to $r = 1.2 - 10 * (0.02 + normalized\ \ IB)$.

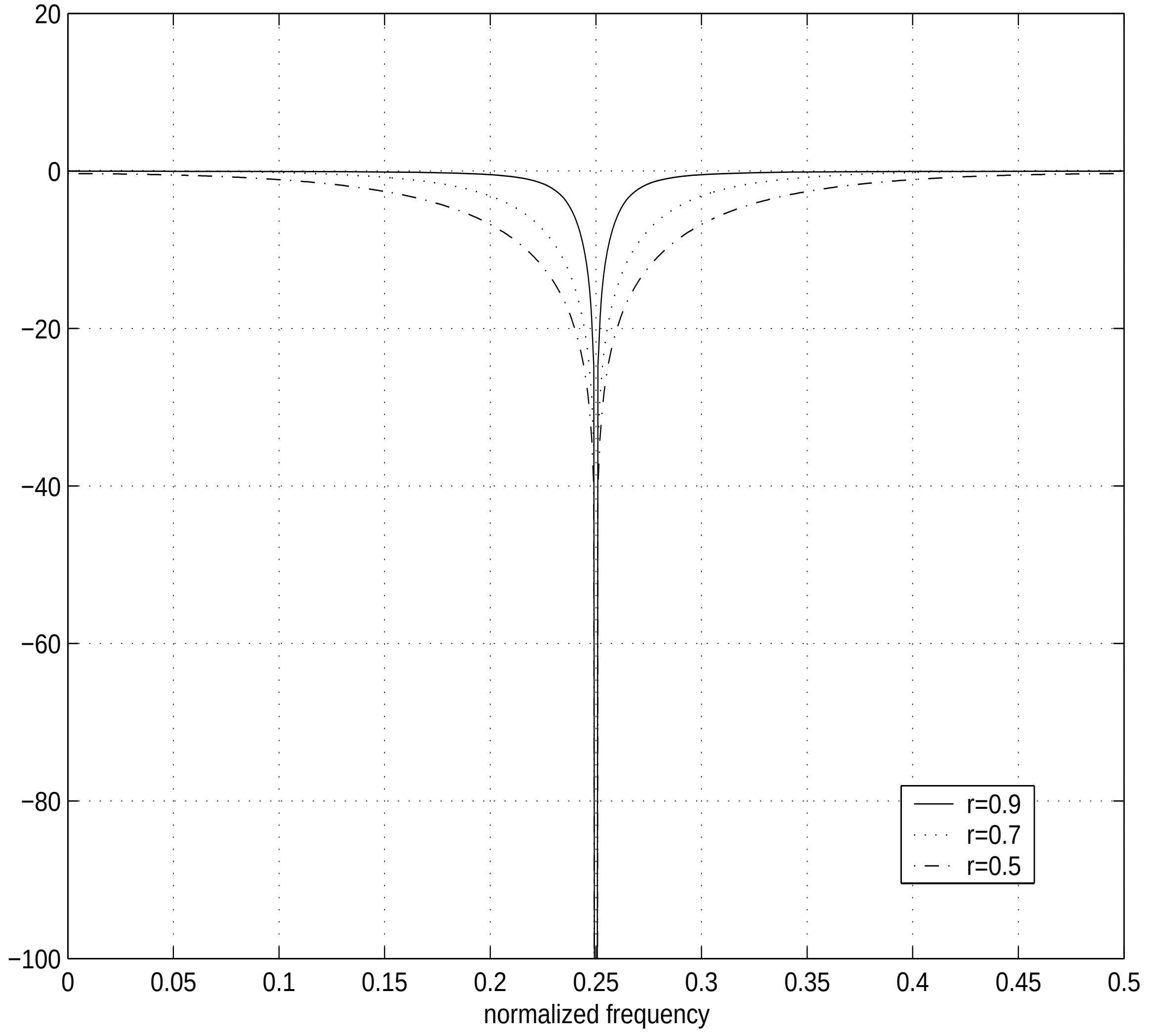


Figure 1. Frequency response characteristics of three notch filter with three different bandwidths at $\pi/2$.

The magnitude response $|H(\omega)|$ of the filter in Eq. 3-12 is plotted in Fig. 1 for $r = 0.75, r = 0.85, and\ r = 0.95$ with $\omega_0 = {}^{\pi}\!/_{2}$. We have used MathWorks Matlab software for the experimental simulation. For the constraints on pole-zero placement of the transfer function, Eq. 3-12, the zeros lie on the unit circle in order to completely null the desired frequency, and the poles lay on the same radial line at a distance r from the origin. The parameter r determines the bandwidth of the notch and as $r \to 1$ the transfer function would describe an ideal notch filter.

Orfanidis modified Eq. 3-12 further to control the notch depth of the filter [23]. The modified transfer function is,

$$H(z,a) = Gain\,\frac{1 - 2\cos\omega_0 z^{-1} + z^{-2}}{1 - 2Gain\cos\omega_0 z^{-1} + (2Gain - 1)z^{-2}} \tag{3-13}$$

where $Gain = \dfrac{1}{1 + \tan\left(\dfrac{BW}{2}\right)\dfrac{\sqrt{1 - a^2}}{a}}$,

where a is the depth control parameter and BW is the notch bandwidth at -3dB.

We can adaptively adjust the notch depth with respect to IB and IP information. We computed a time-dependent adaptive notch depth coefficient a with Eq. 3-13 and replaced BW with IB.

$$a = \frac{P(t)}{[\max(P(t))]^2} - \frac{P(t)}{\max(P(t))} + 1, \qquad 0 \le a \le 1 \tag{3-14}$$

where $P(t)$ is the time marginal distribution (square root of the instantaneous power) of the pathological tremor.

Fig. 2 illustrates the filter characteristics of the second order IIR notch filter with three different notch depth, $a = 0.85, a = 0.95, \textit{and } a = 1.00$ with $\omega_0 = \pi/2$. A second order linear time-varying (LTV) IIR notch filter is implemented via the method of Huang and Aggarwal [21] (which is a sliding block implementation of successive LTI filters). To eliminate the phase distortion that is common in IIR filters, the filter was implemented using forward-backward filtering [22]. Therefore, the output of the second order IIR notch filter in this implementation is the fourth order IIR notch filter.

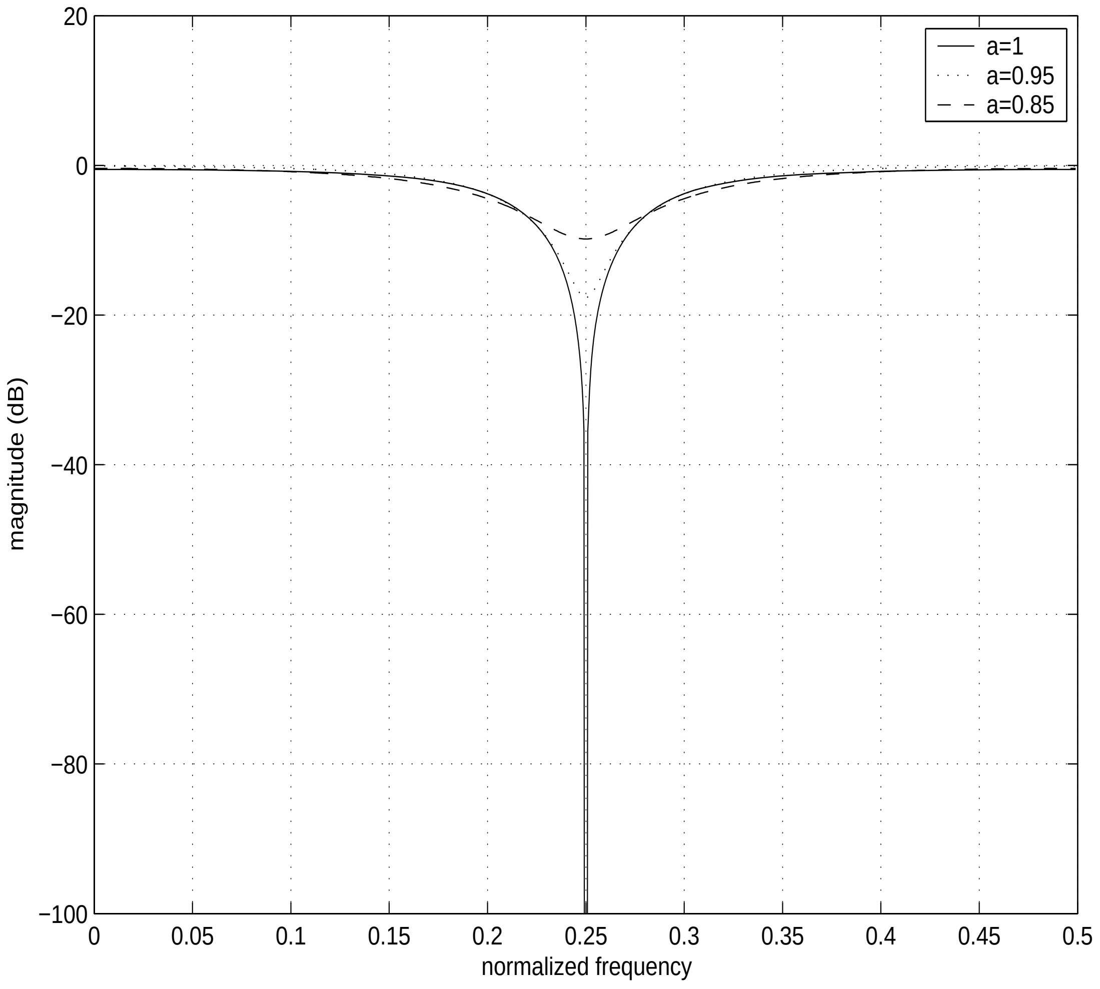


Figure 2: Frequency response characteristics of three notch filter with three different depths at $\pi/2$.

3.3 DEMONSTRATION OF TIME-VARYING NOTCH FILTER

In this section, we will show how the pathological tremor noise in the joystick signal of an electric powered wheelchair is removed by a time-varying notch filter with IF, IB, and IP information. From the real joystick signal with tremor, we will compare the performance between the time-varying notch filter and the current suppression method which is a low-pass filter with 3 Hz cutoff.

3.3.1 Test signal and local parameters (IF, IB, IP)

Generally, the amplitude of pathological tremor noise gradually increases and then decreases. Depending on the intended movement (especially, direction) of the joystick, the tremor signal exists for a short period of time and dies. On the other hand, the range of Cerebral Palsy (CP) tremor frequencies is from 2 to 6 Hz constantly or the frequency may change monotonically. To model CP tremor, we summarize the characteristics of the pathological tremor from the previous studies [1], [2], [7], [8], [10], [11], [12], and [13],

- Roughly sinusoidal movement in 2 to 6 Hz range.
- The amplitude of CP tremor is significantly stronger than an intended movement.
- CP tremor occurs with the intended movement.

Therefore, it is reasonable to model the life of tremor noise with a Gaussian shape of amplitude envelope and a second order of linear frequency modulation.

We assume that our proposed test signal of CP tremor noise is

$$z(t) = A\exp\left[-\alpha\frac{(t-t_0)^2}{2}\right]\cos\left(\beta\frac{t^2}{2}+\omega_0 t\right) \tag{3-15}$$

where A is a constant, α is a variance of Gaussian envelope, t_0 is a time delay and β is the chirp rate of the tremor. When α is close to zero, $z(t)$ has a constant amplitude envelope and when β is close to zero, z(t) becomes a tone signal (just one constant frequency).

We need to use an analytic signal (complex representation) of the real signal for the joint time-frequency distribution (TFD). The analytic signal of $z(t)$ is obtained by using Hilbert transformation and rewritten in Eq. 3-16.

$$s(t) = A\exp\left[-\alpha\frac{(t-t_0)^2}{2}+j\left(\beta\frac{t^2}{2}+\omega_0 t\right)\right]. \tag{3-16}$$

The spectrum of the test signal is obtained without difficulty,

$$S(\omega) = \frac{A(\pi/\beta)^{\frac{1}{4}}}{\sqrt{\alpha - j\beta}}\exp\left[-(\alpha + j\beta)\frac{(\omega-\omega_0)^2}{2(\alpha^2+\beta^2)}\right] \tag{3-17}$$

$$|S(\omega)|^2 = \frac{A^2}{\sqrt{\sqrt{\pi}\,(\alpha^2+\beta^2)}}\exp\left[-\alpha\frac{(\omega-\omega_0)^2}{\alpha^2+\beta^2}\right]. \tag{3-18}$$

Calculated spectrogram of the test signal with Gaussian window is

$$P_{SP}(t,\omega) = \frac{P(t)}{\sqrt{2\pi\sigma^2_{\omega|t}}} \exp\left[-\frac{\left(\omega - \langle\omega\rangle_t\right)^2}{2\sigma^2_{\omega|t}}\right] = \frac{P(\omega)}{\sqrt{2\pi\sigma^2_{t|\omega}}} \exp\left[-\frac{\left(t - \langle t\rangle_\omega\right)^2}{2\sigma^2_{t|\omega}}\right]. \qquad (3\text{-}19)$$

For simplicity, we set $\beta = 0$ in this demonstration from now. That is, the test signal has a constant frequency which is the case of a single constant tremor frequency. The normalized test signal of Eq. 3-16 and its corresponding energy density spectrum are shown in Fig. 3. Next, we estimated the IF, IB, and IP of the test signal from Spectrogram by using the methods in the previous section and drew them in Fig. 4, on top of the time-frequency distribution.

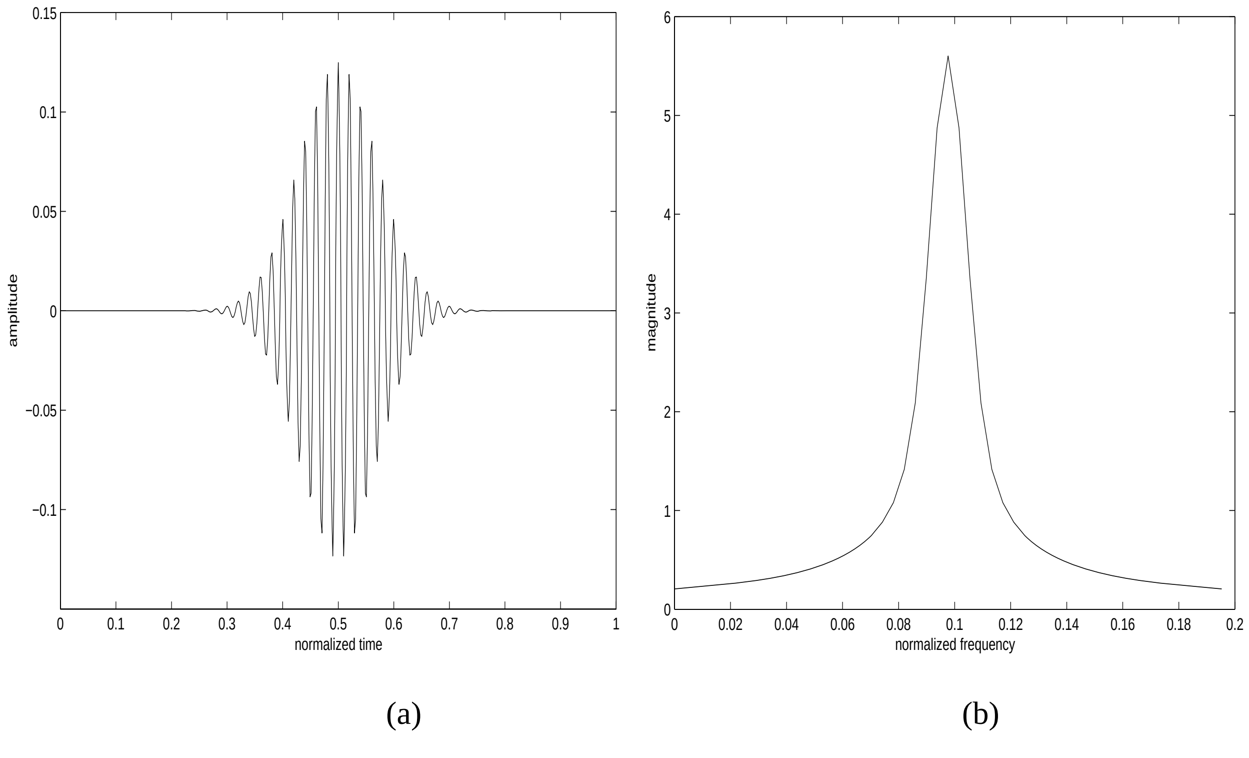
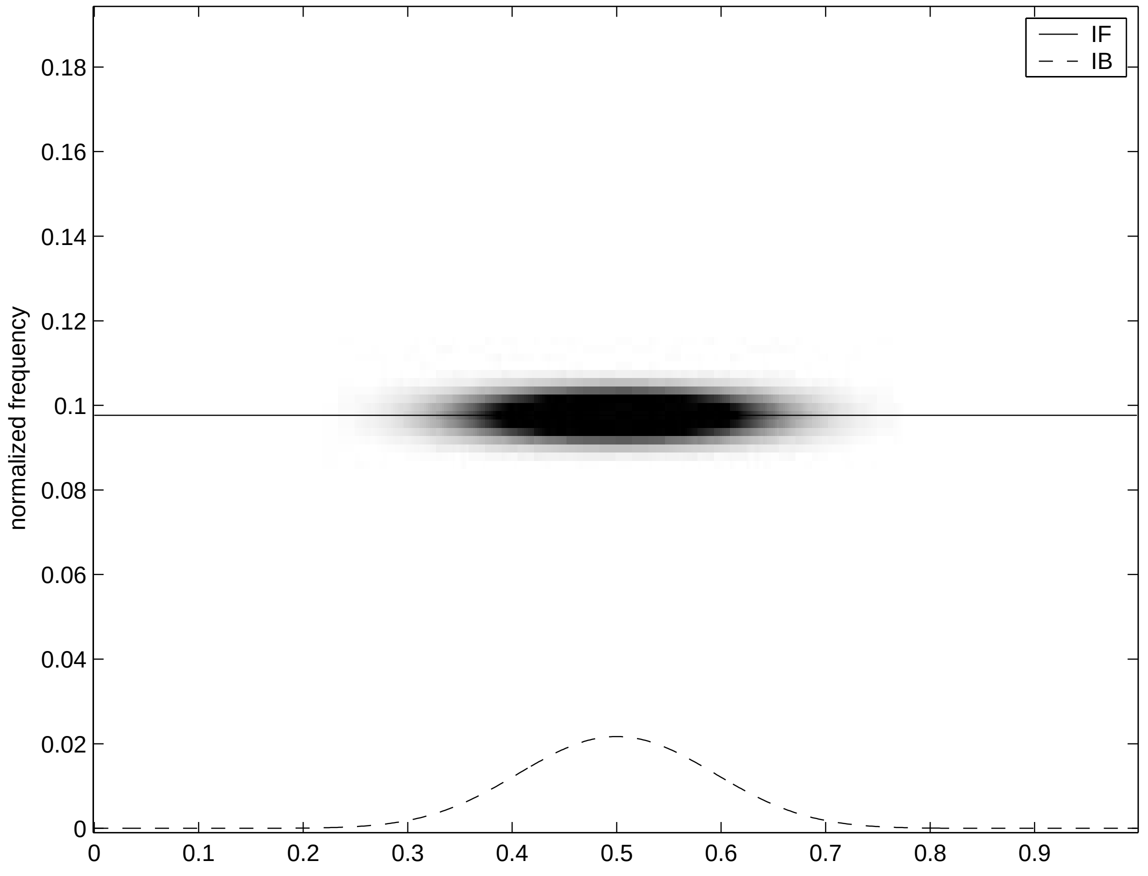
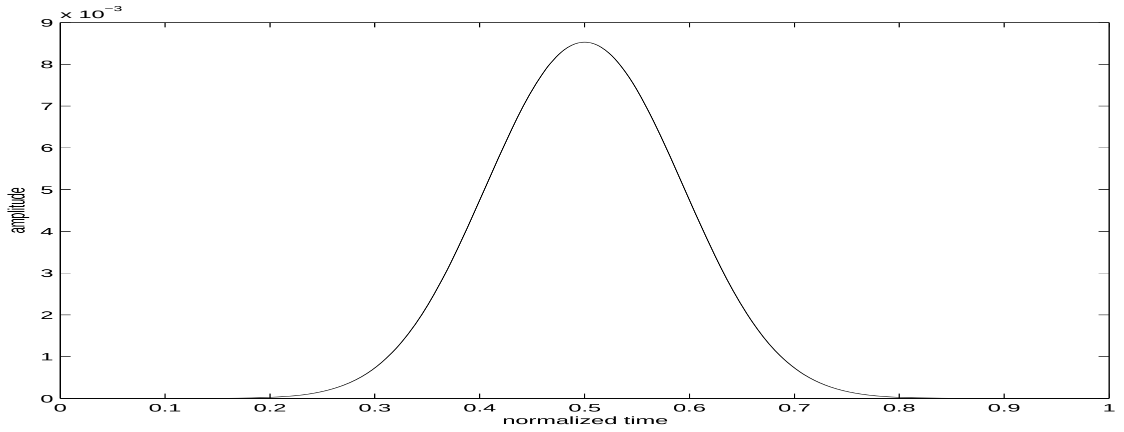


Figure 3: (a) Normalized waveform of the test signal and (b) its corresponding power spectral density ($\alpha = 100$, $\beta = 0$, and $\omega_0 = 2\pi \cdot 50$ with 512 is the size of data and FFT point).



(a)



(b)

Figure 4: (a) TFD of the test signal with IF (solid line) and IB (dotted line) and (b) time-marginal distribution, IP.

3.3.2 Tuning notch filter with IF, IB, and IP

With the equations in the section 3.2, the notch width and depth control parameters, r and a respectively, are calculated by the values of estimated IF, IB, and IP, therefore the shape of the notch is adjusted by them. Fig. 5 shows how those parameters change with respect to time. At time, $t = 0.5$, those parameters generate the widest and deepest notch that matches with the test signal.

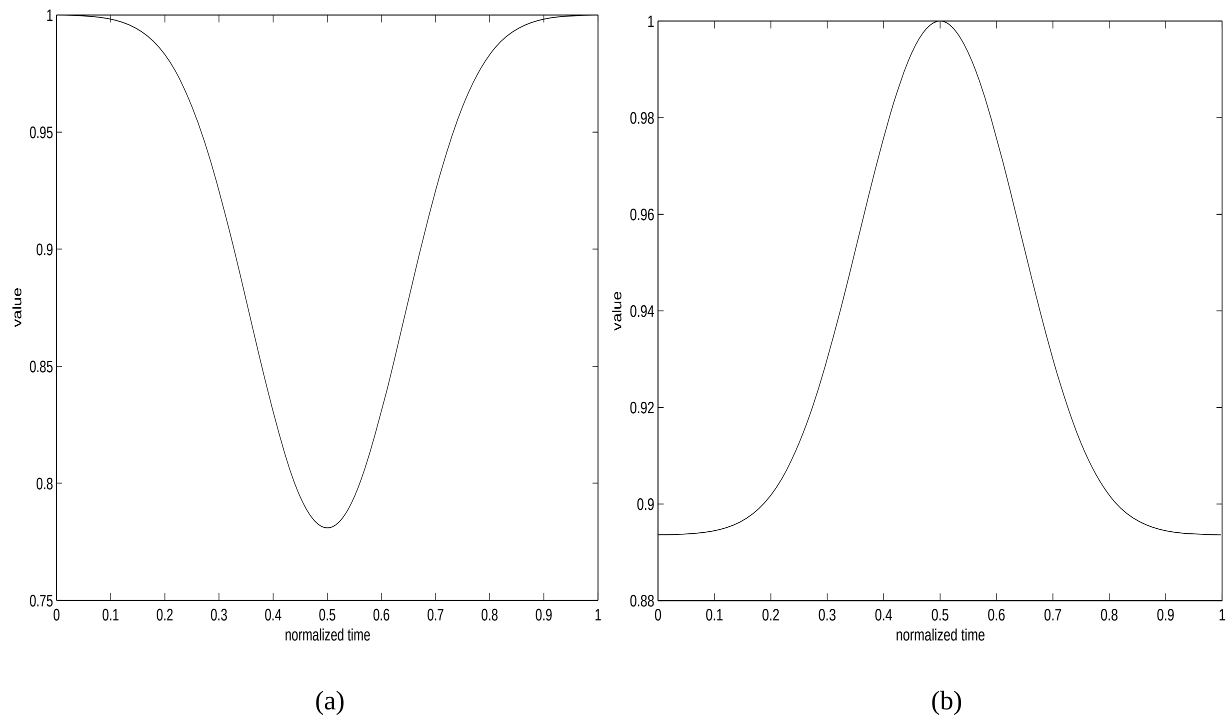


Figure 5: Time-varying (a) notch-width control parameter, r , and (b) notch-depth control parameter, a , for the test signal.

Now, we are able to obtain the adaptive notch filter for the test signal. Fig. 6 shows the characteristics of the time-varying notch filter at three different time points, $t = 0.1, 0.3, \textit{and } 0.5$.

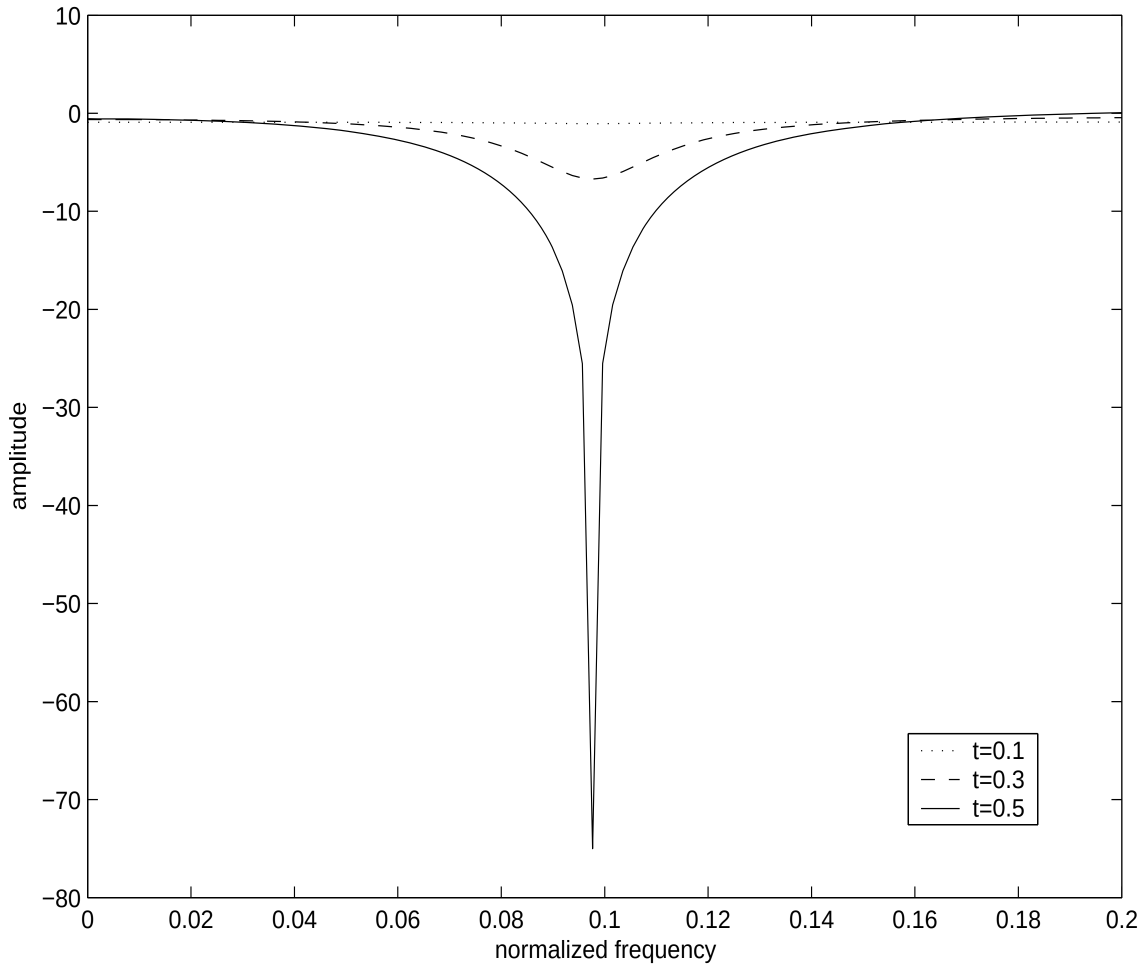


Figure 6: Characteristics of notch for the test signal at three different time

After we apply the notch filter to the test signal, we are able to suppress the signal nicely. Fig. 7 shows the final result of proposed filtering method for the test signal. In this section, we have demonstrated how the time-varying notch filter is constructed in detail. We also showed the importance of IF, IB, and IP in the time-varying notch filter.

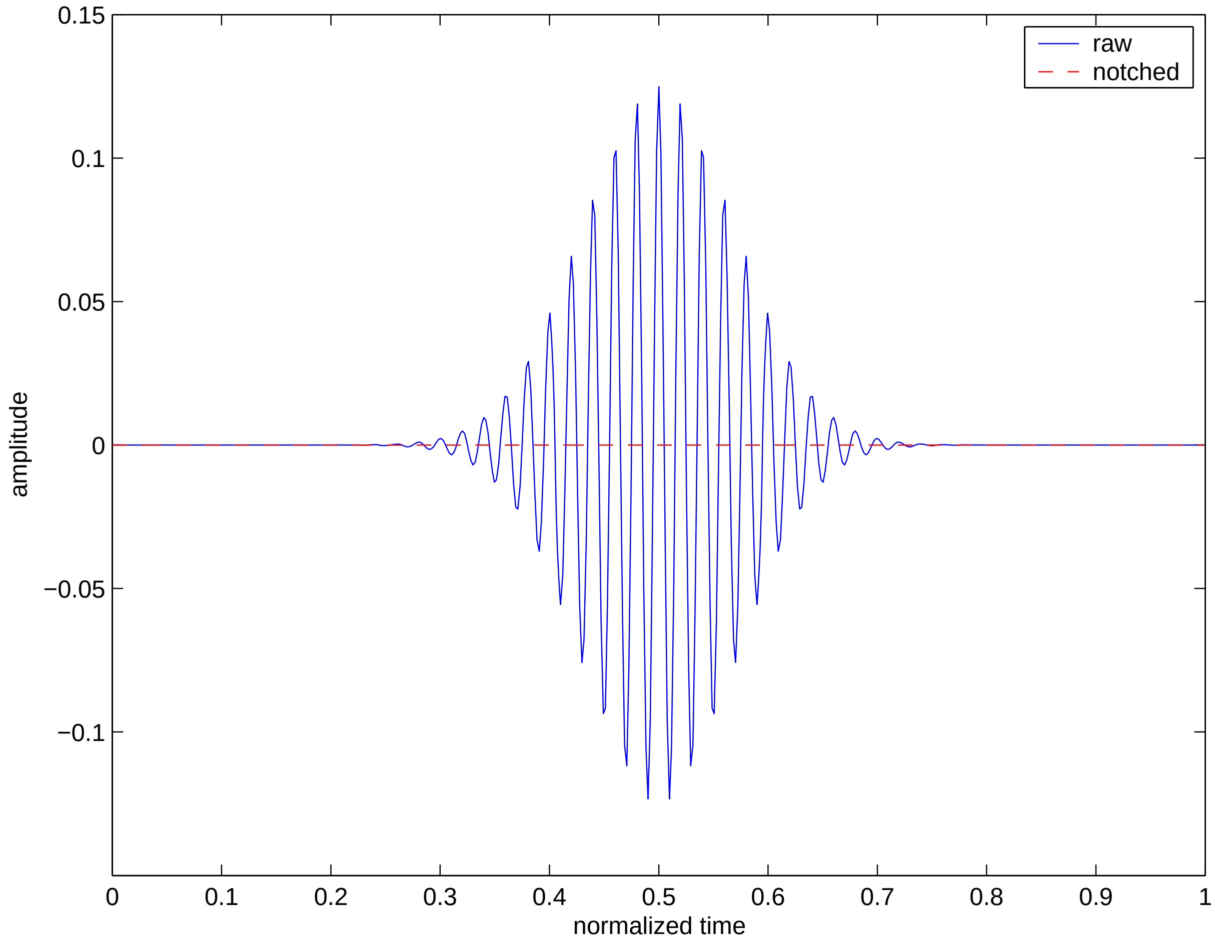


Figure 7: Final result of proposed filtering method for the test signal.

3.4 PRELIMINARY JOYSTICK DATA

In this section, we applied the time-varying notch filter which is tuned by IF, IB, and IP obtained from joint time-frequency distribution to real joystick signals containing Cerebral Palsy (CP) tremor noise and compared the results of the notch filter with the results of the commonly used low-pass filter (LPF) with 3 Hz cutoff.

First, we will briefly describe the experimental background and setting for the preliminary data. All experiments were performed in the Human Engineering Research Laboratories (HERL). Originally, the joystick signals were captured to compare the control signals generated during simple driving tasks using a standard position sensing joystick and a force sensing joystick (FSJ) with two distinct control algorithms [28]. The FSJ algorithms are a simple constant proportional gain with a LPF for both speed and direction, and a variable gain algorithm with a low-pass digital filter. Seventeen EPW users with cerebral palsy and upper extremity tremor and athetosis performed the test with Quickie P300 (Sunrise Medical, LTD) to evaluate these engineering methods for improving the control of an EPW.

The joystick information was collected at 322 Hz sampling rate while each subject drove to the center of nine "targets" arranged at various angles and distance from a common starting point (shown in Fig. 8). Fig. 8 is a scale drawing of the testing room and provides the dimensions and locations of target. In this experiment, an 8-bit counter is used to measure the displacement of the joystick and the range of the joystick movement is ± 54 counts from the center point, 113 counts. The starting joystick displacement of 113 counts indicates the joystick is at the center position. When the chair speeds up and down, the displacement counts increase and decrease respectively.

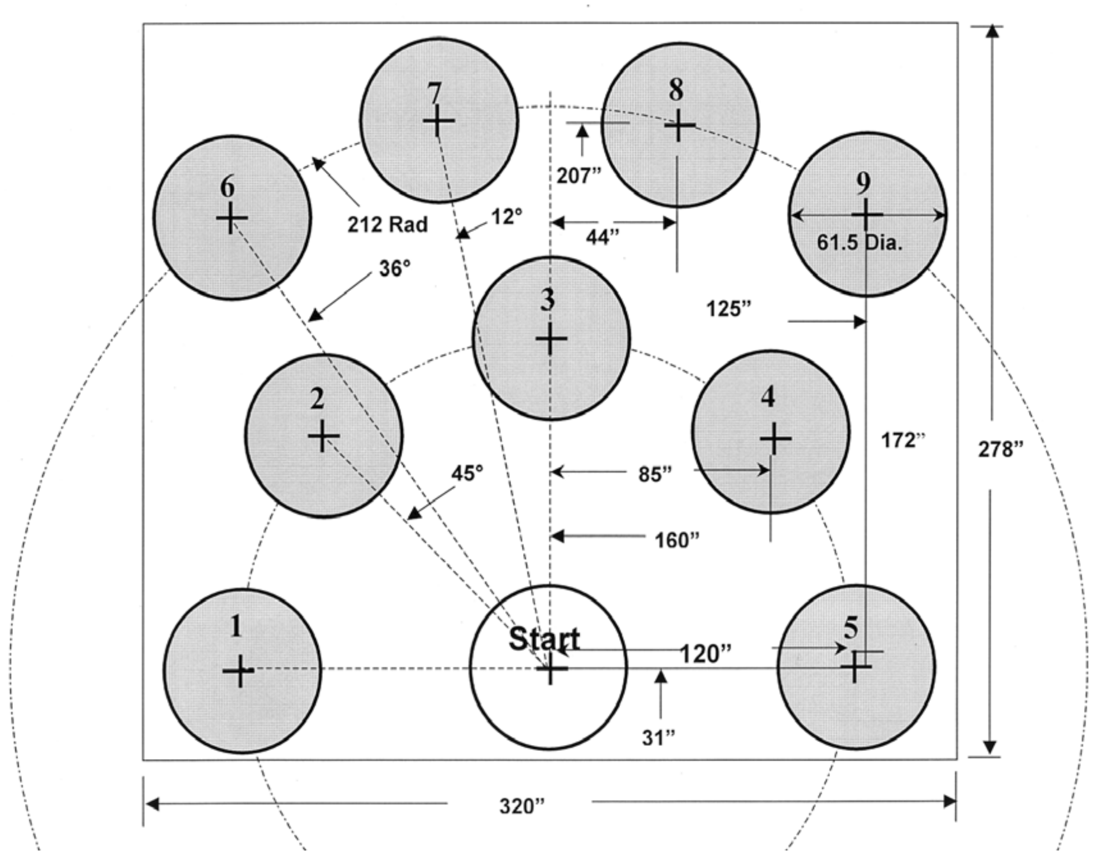


Figure 8: Experiment settings (nine targets)

For our preliminary data analysis, we selected raw data from subject number 11, the third trial, and speed displacement (y-axis) because it has a clear tremor behavior (athetoid movement) and it is easy to assume the intended movement. Fig. 9 shows that the joystick signal is corrupted severely by a pathological tremor which creates a serious problem for driving an electric powered wheelchair. For this particular data, it is reasonable to use the LPF with a 1 Hz cut-off frequency to obtain the assumed intended movement since we did not observe any abrupt driving behavior except that the chair sped up and down slowly to get to the target 7. Fig. 9 shows the raw joystick signal with the assumed intended movement.

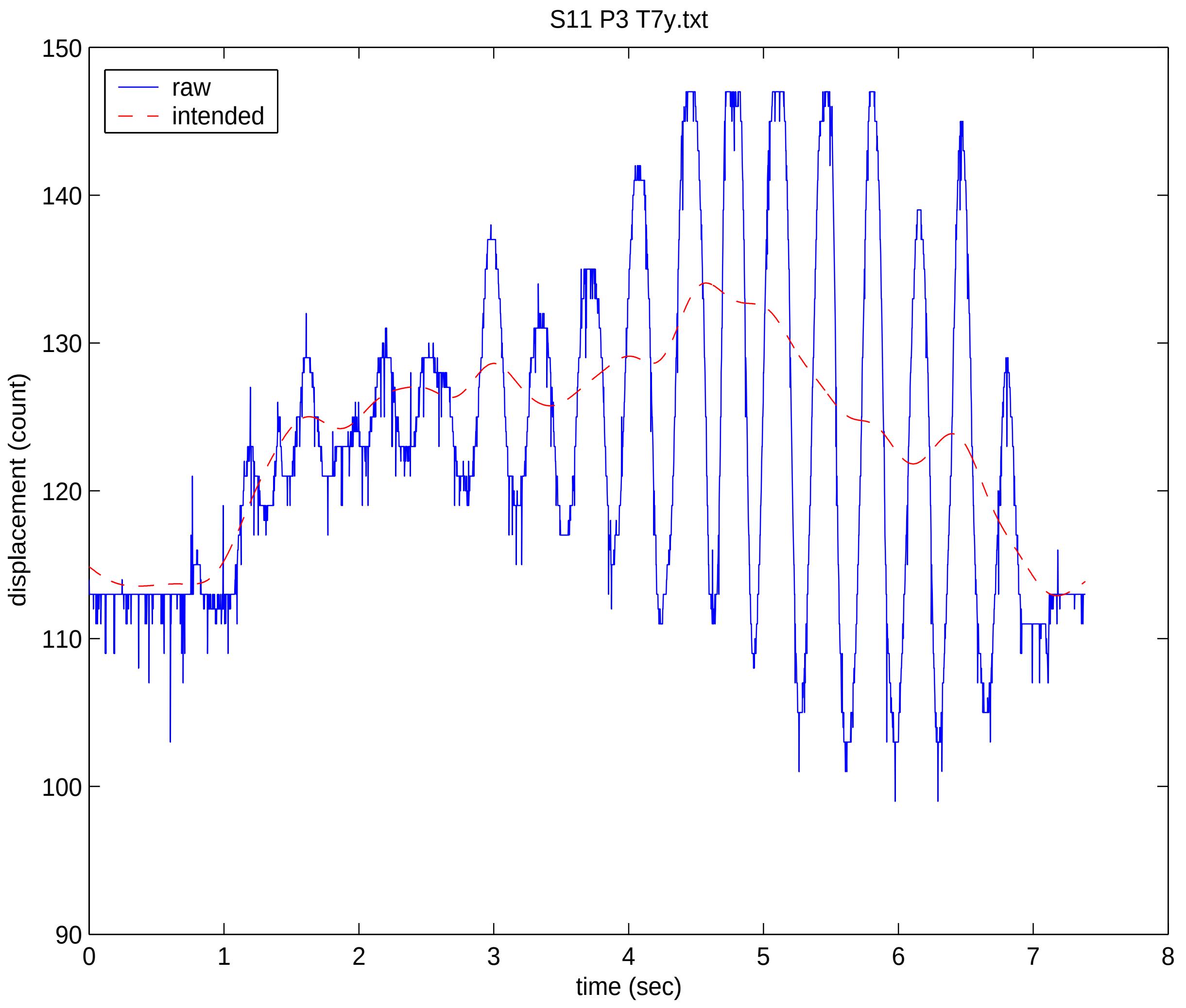


Figure 9: Example of test compromised by CP patient: the wheelchair driving (S11 P3 T7y.txt is the data file name where S11 is a subject number 11, P3 is a trial number 3 , and T7y is the joystick displacement in y-axis for a target number 7) and dash line is an intended movement.

From the figure, we can see approximately little less than 3 Hz of CP tremor frequency with high peak to peak variation and mainly in the intervals of 3-7 seconds. Therefore, if we apply the 3 Hz of a low-pass filter to the raw data, there is still a high chance that some unwanted tremor movement will remain. As we learned about the characteristics of CP tremor in chapter two, CP tremor is roughly a high powered sinusoidal movement and is in the frequency range of 2-6 Hz. Therefore, applying a band-pass filter (BPF) with a 1-7 Hz pass-band is a reasonable choice to see the clear picture of the CP tremor without interfering with the intended movement. The outcome of the band-pass filtered joystick signal is shown in Fig. 10 and its Spectrogram is shown in Fig. 11. From Fig.11, we can assume that CP tremor occurs at 3 Hz frequency and mainly in the intervals of 3-7 seconds.

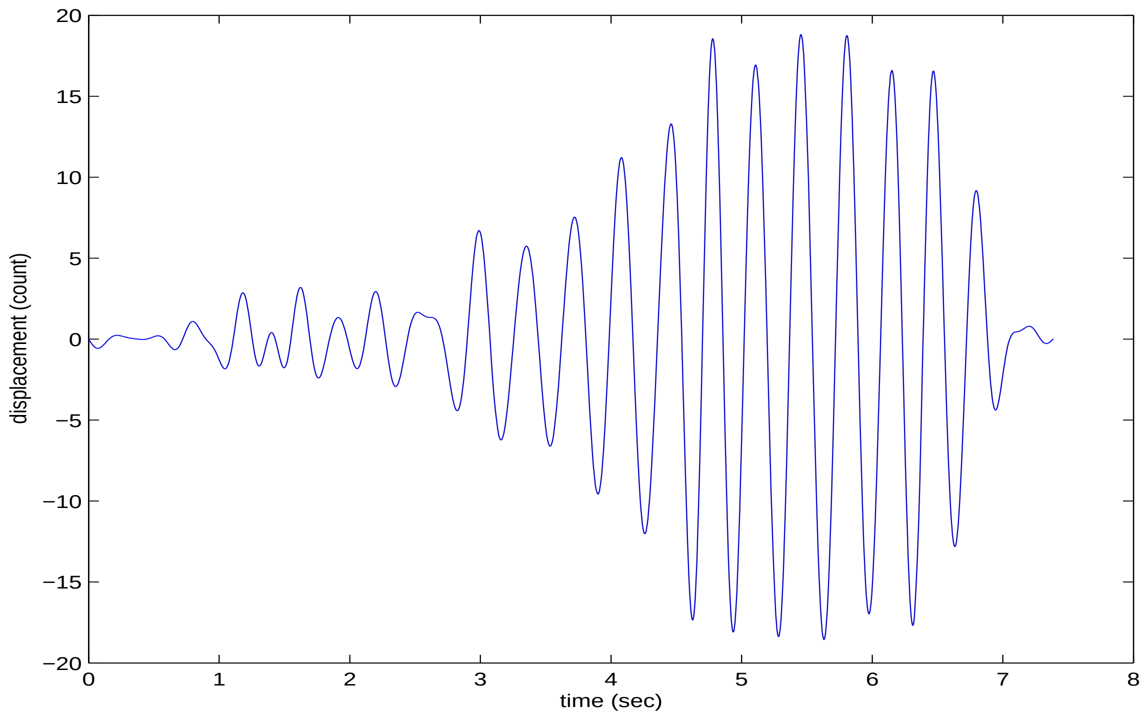


Figure 10: Band-pass filtered joystick signal.

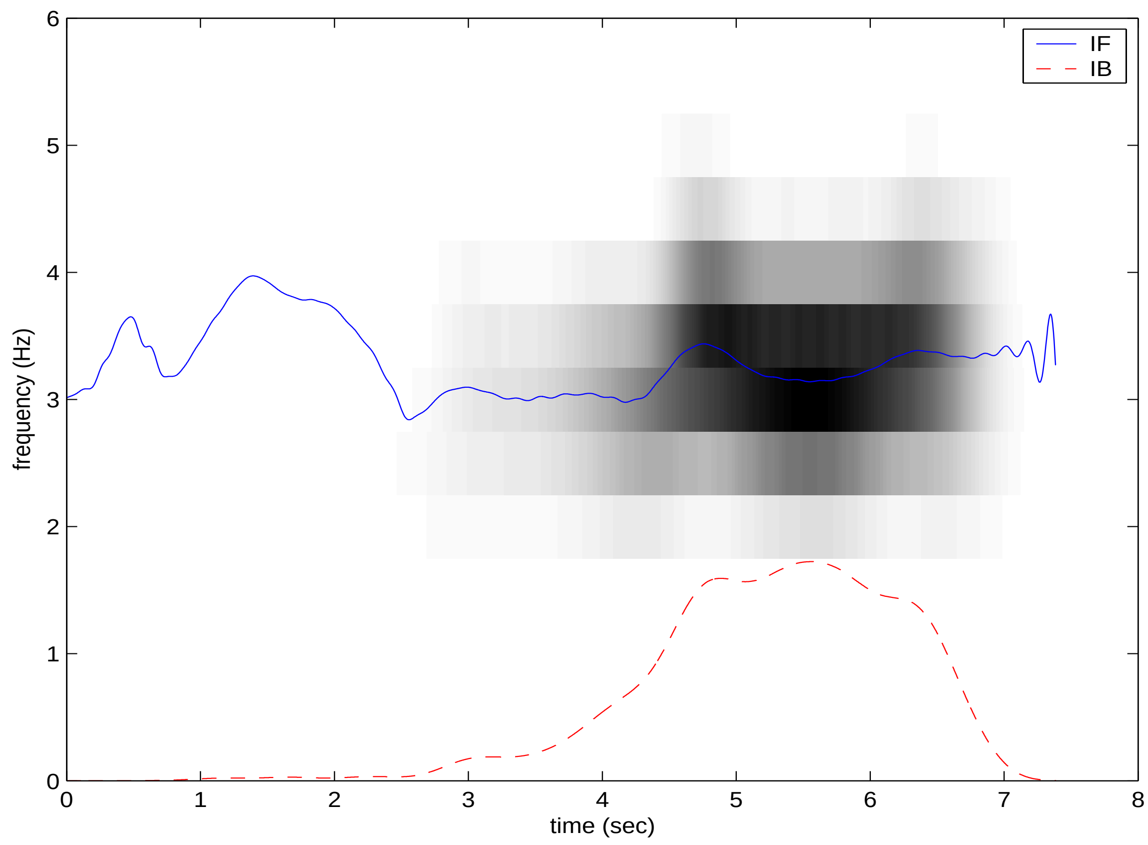


Figure 11: Spectrogram of CP tremor signal with IF and IB.

With the band-pass filtered signal, we obtained the estimated IF and IB of the test signal from Spectrogram by using the methods in the section 3.1 (shown in Fig. 11) and calculated the notch width and depth control parameters, r and a respectively and designed the time-varying adaptive notch filter. After we applied the proposed suppression method like we did to the test signal, we are able to filter the tremor noise out effectively and the result is shown in Fig. 12. Additionally, Fig. 12 shows the visual comparison between the proposed method and LPF with a 3 Hz cutoff frequency.

The output of the LPF shows that the filter did not filter the 3 Hz tremor noise completely because some portion of the unwanted CP tremor noise is still present. If we narrow the pass-band of the LPF, the oscillation is minimized, but there is a high chance of losing desired information. On the other hand, although the proposed filter did not suppress the oscillating noise completely in the entire area, there is a significant improvement in the high-powered noise area where it creates a major problem for driving an electric powered wheelchair with a joystick.

To have a performance comparison between the proposed method and the LPF, we calculated mean square error (MSE) for the results of both suppression methods. MSE defined as

$$MSE = E\{(y - x)^2\} \tag{3-16}$$

MSE for the LPF is 3.5807 counts and MSE for the proposed method is 0.8095 counts. Therefore, from MSE numbers, we can conclude that the proposed method is more effective than the LPF for this joystick signal. However, depending on where CP tremor noise and an intended movement are detected in frequency domain and how much power ratio they have with respect to each other, the performance of those suppression methods will be changed. We will deal with this problem in next section.

In conclusion, experimental results show that the proposed suppression method, time-varying notch filter with IF, IB, and IP, is able to reduce the tremor signals effectively, while retaining the features of the desired joystick signal for the off-line experiment. Therefore, instantaneous frequency, instantaneous bandwidth, and instantaneous power can be good candidates for the suppression process as well as for the characteristics of pathological tremor noise.

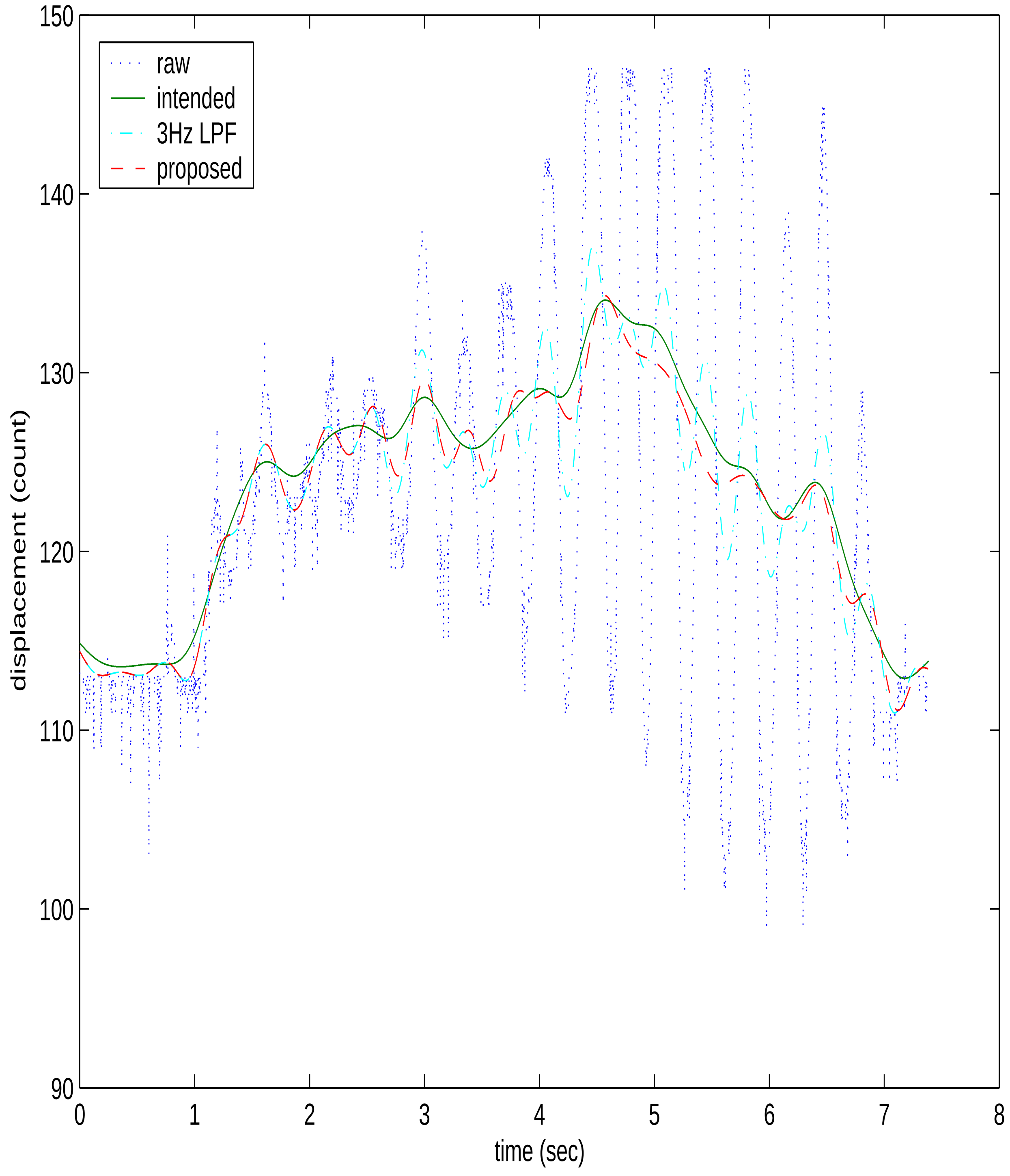


Figure 12: The results for the proposed suppression method and LPF with 3 Hz cutoff frequency (dot line (..) is a raw joystick signal, solid line (-) is an assumed intended movement, dash-dot (-.) is a LFP (3 Hz) signal and dash-dash line (--) is a notched signal with IF, IB, and IP.

3.5 PROBLEMS OF TIME-VARYING NOTCH FILTER

We introduce potential problems associated with the use of a time-varying notch filter with TFD method to suppress CP tremor in this section. We will focus on the question: When does our proposed CP tremor suppression system break down? We need to be wary of two major cases: (1) TFD does not provide us good IF and IB estimations from TFD plane, possibly because of unexpected high noise or very low energy CP tremor in the joystick system (unlikely), and (2) intended movement and CP tremor frequencies occur very close to each other, such that we face severe performance degradation in the CP tremor suppression system. To address these potential problems, we need to set constraints for our CP tremor suppression system.

3.5.1 When noise level is very high or CP tremor power is low

The intended movement data is corrupted by both additive noise and CP tremor in the EPW joystick system. The output signal from the joystick is therefore modeled as

$$J(t) = S(t) + T(t) + N(t) \tag{3-17}$$

where $S(t)$ is an intended movement signal, $T(t)$ is an unwanted tremor frequency, and $N(t)$ is zero-mean additive white Gaussians noise (AWGN) with variance σ^2.

Depending on the power of intended movement of the EPW joystick, CP tremor, and noise, the success of IF and IB estimation is decided based on the use of the peaks and standard deviation from the TFD plane as IF and IB estimators, respectively. We examine the effects of AWGN and CP tremor power levels on IF estimation. For computer simulation settings, we use simple sinusoidal signals for CP tremor noise $T(t)$ with 5 Hz frequency and intended movement signal $S(t)$ with 2 Hz frequency, which are distinctly different. Therefore, real IF of CP tremor noise is 5 Hz. We run the simulation with several different combinations of the intended movement signal-to-tremor noise power ratio (STR) and the intended movement signal-to-AWGN power ratio (SNR). An intended movement-to-tremor noise power ratio STR defined as,

$$STR(dB) = 10\log\left(\frac{P_S}{P_T}\right) \tag{3-18}$$

where P_S is the intended movement signal power and P_T is the tremor noise power.

Table 1: MSE test between real IF and estimated IF with different combinations of SNR and STR.

SNR (dB) at	MSE (STR (dB) = -18.2)	STR (dB) at	MSE (SNR (dB) = -7.2)
9.8518	0	-32.0712	0
-6.9718	0	-25.1397	0
-23.0539	0	-18.2082	0
-29.7279	0	-9.0453	0
-35.8550	0.58	-1.0603	1.2154
-39.5341	72.1670	4.8176	2.998
-45.6509	70.88	20.912	2.998

We calculate MSE between real IF and estimated IF and the results are shown in Table 1. First, we use a fixed STR of -18.2 dB and vary the value of SNR from 9.8518 dB to -45.6509 dB to see the effect of the noise on IF estimation, that shows on first two columns of Table 1. And then we use a fixed SNR of -7.2 dB and vary the value of STR from -32.0712 dB to 20.912 dB, that shows on next two columns of Table 1. From the experiment, we can see the effects of SNR and STR on IF estimation. High noise and high intended movement signal powers create problems on IF estimation. The same results are found in IB estimation. Incorrectly-estimated IF and IB have a detrimental effect on the performance of tremor suppression system. To avoid this problem, we may set power ratio limitation on our suppression system such as a shared control. For example, a time-varying notch filter is applied in the case of high powered tremor case, otherwise LPF is applied.

3.5.2 When an intended movement and P tremor frequencies occur very close to each other

Depending on the locations of intended movement frequencies of EPW joystick and CP tremor frequency, the performance of suppression methods can be varied. Generally, intended movement frequencies are lower than CP tremor frequencies, but intended movement frequencies are may also occur very close to or higher than CP tremor frequency. If intended movement frequencies are maintained under a low-pass filter and CP tremor frequency is outside of the low-pass filter band, then the low-pass filter works best. On the other hand, a time-varying notch filter works better than a low-pass filter in cases where the intended movement frequencies are close to or higher than the CP tremor frequency.

Table 2: Comparison between LPF with 3 Hz and time-varying notch filter results when CP tremor occurs at 3 Hz frequency.

Intended Movement Frequency	MSE for Low-pass Filter with 3 Hz	MSE for Time-Varying Notch Filter
1 Hz	0.0086	0.0090
2 Hz	0.0087	0.0100
3 Hz	0.0136	0.0107
4 Hz	0.133	0.0099
5 Hz	0.133	0.0089

For the computer simulation, we use a simple sinusoidal signal with 3 Hz frequency for CP tremor noise and several different sinusoidal signals with 1 to 5 Hz frequencies for an intended movement signal. For additional test settings, it is reasonable to set the intended movement-to-tremor noise ratio (STR) of -16 dB and the intended movement signal-to-AWGN ratio (SNR) of 16 dB since the energy level of CP tremor noise is much stronger than that of an intended movement and AWGN. The simulated results show in Table 2. From Table 2, if intended movement frequency is maintained under 3 Hz, then LPF has better performance results than a time-varying notch filter. Otherwise, a time-varying notch filter is a better tremor suppression method. We can utilize both of the filtering methods to compensate for their individual weaknesses. However, sharing filters for the different cases only works in off-line suppression because the time-delay of a low-pass filter presents a greater problem in real-time practical environments.

4.0 TECHNICAL DEVELOPMENT: ADAPTIVE NOTCH FILTER (ON-LINE)

In this chapter, we introduce a real-time tremor suppression method. Pathological tremor is a non-stationary process exhibiting characteristics of frequency and amplitude modulation. Adaptive algorithms are particularly suited to active tremor canceling since they adjust automatically to such signal changes over time. Our proposed suppression method to the real-time joystick operation is a time-varying adaptive notch filter. Two channels of filtering are used for the two orientations (x and y) of joystick input. We introduce Riviere's Weighted-frequency Fourier Linear Combiner filter (WFLC) as a time-varying adaptive notch filter in our study since it can estimate tremor amplitude as well as tremor frequencies. Riviere and Thakor developed the WFLC and applied it to a computer pen and a hand-held surgical device application [15]. They demonstrated that WFLC was a good choice for tremor suppression problems. The background of WFLC will be shown in the following sections. Since WFLC estimates the notch frequencies directly and not indirectly from the transfer function parameters, we do not need to observe a stability monitoring as long as the value of step size maintains a stable range. In the last section of this chapter, we modified Riviere's WFLC by adding instantaneous bandwidth (IB) information to the algorithm since we had shown a notch filter performed better with IB information from the previous chapter. We demonstrated that the modified WFLC performed better than Rivere's WFLC by using a synthetic signal.

4.1 ADAPTIVE NOISE CANCELING

The most relevant technique for tremor suppression application is adaptive noise canceling. An adaptive noise canceller is a noise filter that self optimizes on-line as it encounters an input signal, adjusting its parameters according to a learning algorithm. The block diagram of the adaptive noise canceller is drawn in Fig. 13. As shown in Fig. 13, the system accepts two inputs: a primary input s_k, containing a desired signal d_k and uncorrelated noise n_k; and a reference input x_k, containing noise n'_k correlated with n_k. Then y_k is subtracted from s_k to yield ε_k. The adaptive process minimizes the mean square value of ε_k, thereby minimizing the mean square error between n_k and y_k, making ε_k an estimate of d_k. Adaptation is typically accomplished using a gradient descent algorithm such as the popular least mean square (LMS) algorithm. The reference input is processed by an adaptive filter that automatically adjusts its own impulse response through LMS that responds to an error signal dependent on the filter's output. In noise canceling systems the practical objective is to produce a system output, ε_k, that is a best fit in the least squares sense to the signal d_k. The objective is accomplished by feeding the system output back to the adaptive filter and adjusting the filter through an adaptive algorithm to minimize the total system output power. Therefore, the system output serves as the error signal for the adaptive process in noise canceling system.

The output is

$$\varepsilon_k = d_k + n_k - y_k \tag{4-1}$$

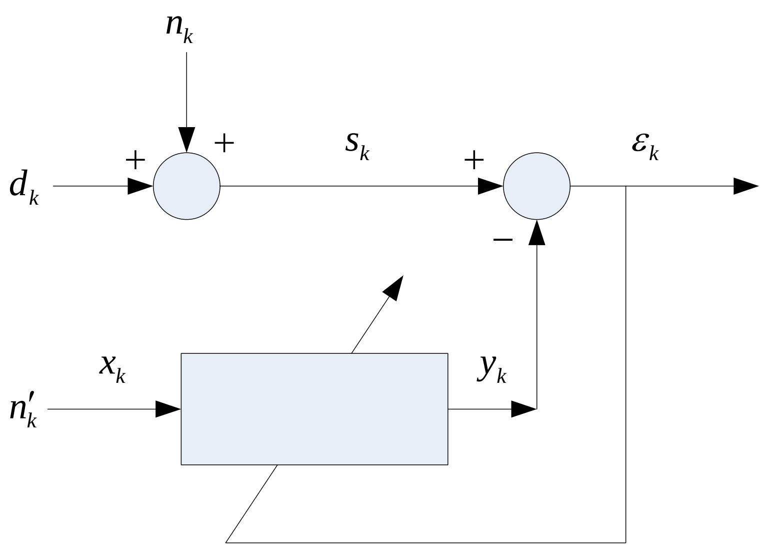


Figure 13: Adaptive noise canceller. A desired signal, d_k is corrupted by a noise, n_k. The adaptive filter takes in a noise n'_k, correlated with n_k, and outputs an estimate y_k of n_k, which is subtracted from the primary input, s_k. This yields ε_k, which is an estimate of the desired signal, d_k.

By squaring Eq. 4-1, we obtain

Adaptive
Filter

$$\varepsilon_k^2 = d_k^2 + (n_k - y_k)^2 + 2d_k(n_k - y_k) \tag{4-2}$$

Taking expectation of both sides of Eq. 4-2, and realizing that d_k is uncorrelated with n_k and with y_k**,** yields

$$E[\varepsilon_k^2] = E[d_k^2] + E[(n_k - y_k)^2] \tag{4-3}$$

The signal power $E[d_k^2]$ will be unaffected as the filter is adjusted to minimize $E[\varepsilon_k^2]$ Therefore, the minimum output power is when

$$E_{\min}\left[\varepsilon_k^2\right] = E\left[d_k^2\right] + E_{\min}\left[\left(n_k - y_k\right)^2\right] \tag{4-4}$$

When the filter is adjusted so that $E\left[\varepsilon_k^2\right]$ is minimized, $E\left[\left(n_k - y_k\right)^2\right]$ is therefore also minimized. The filter output y_k is then a best least squares estimate of the noise n_k. Moreover, when $E\left[\left(n_k - y_k\right)^2\right]$ is minimized, $E\left[\left(\varepsilon_k - d_k\right)^2\right]$ is also minimized, since

$$\left(\varepsilon_k - d_k\right) = \left(n_k - y_k\right) \tag{4-5}$$

The output ε_k will generally contain the signal d_k plus some noise. From Eg. 4-5, the output noise is given by $\left(n_k - y_k\right)$. Since minimizing $E\left[\varepsilon_k^2\right]$ minimizes $E\left[\left(n_k - y_k\right)^2\right]$, minimizing the total output remains constant, minimizing the total output power maximizes the output signal to noise ratio. We see from Eg. 4-4 that the smallest possible output power is $E_{\min}\left[\varepsilon_k^2\right] = E\left[d_k^2\right]$. When this is achievable, $E\left[\left(n_k - y_k\right)^2\right]$ = 0. Therefore, $y_k = n_k$ and $\varepsilon_k = d_k$. In this case, minimizing output power causes the output signal to be perfectly free of noise. Notice that in this application, the error signal actually converges to the input data signal, rather than converging to zero.

This adaptive noise canceling scheme requires a reference signal. In many human-machine control applications, obtaining a reference signal that contains tremor is not convenient. However, there are a number of ways to resolve this difficulty. For periodic interference, one such method is to generate a reference input via a tapped delay line that receives the primary input, delayed by some amount. This adaptive filter structure therefore attempts a linear prediction of the current noise value based on past value, and is known as an adaptive predictor. Implementing this type of filter for tremor canceling in control applications is problematic for two reasons. Little is known about the human voluntary motion so it is difficult to determine a suitable value for the tapped delay. A more significant drawback is that the system models the interference as a linear autoregressive process. Since pathological tremors are non-linear process, the linear prediction is therefore unlikely to yield a satisfactory estimate of the tremor, particularly when delayed by a potentially large value.

4.2 FLC

The roughly sinusoidal nature of tremor makes it well-suited to a Fourier representation. The Fourier linear combiner (FLC) is an adaptive filter that forms a dynamic truncated Fourier series model of an input signal [30]. The series model

$$y_k = \sum_{r=1}^{M} \left[a_r \sin(r\omega_0 k) + b_r \cos(r\omega_0 k) \right] \tag{4-6}$$

where the Fourier coefficients, a_r and b_r, are the adaptive filter weights.

The FLC operates by adaptively estimating the Fourier coefficients of the model according to the LMS algorithm. The block diagram of the FLC is presented in Fig. 14 and the algorithm is as follows,

$$\begin{gathered} x_{r_k} = \begin{cases} \sin(r\omega_0 k), & 1 \le r \le M \\ \cos[(r-M)\omega_0 k], & M+1 \le r \le 2M \end{cases} \\ \varepsilon_k = s_k - W_k^T X_k \\ W_{k+1} = W_k + 2\mu X_k \varepsilon_k \end{gathered} \tag{4-7}$$

where $W_k = \left[\omega_{1_k}, \omega_{2_k}, \quad \cdots \quad, \omega_{2M_k}\right]^T$ is the adaptive weight vector, s_k is the input signal, M is the number of harmonics in the model, and μ is an adaptive gain parameter.

The adaptive weight vector, W_k, generates a linear combination of the harmonic orthogonal sinusoidal components of the reference input vector, X_k. The FLC effectively estimates and cancels periodic interference of known frequency. Several features of the FLC are useful for canceling quasi-periodic interferences such as tremor.

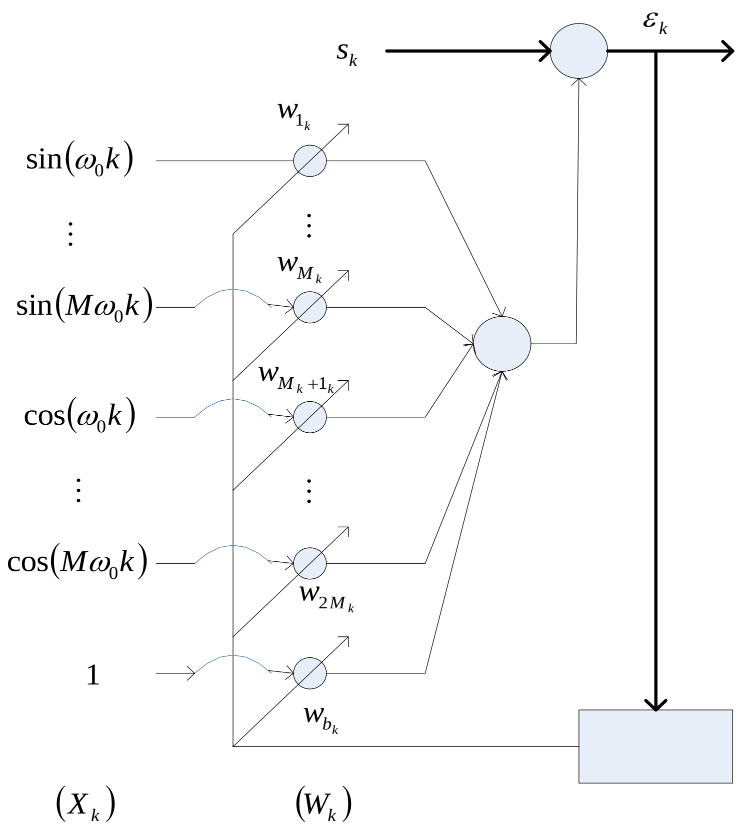


Figure 14: The Fourier linear combiner. The FLC adaptively creates a dynamic Fourier series model of an input signal that can be used to cancel a quasi-periodic interference provided the fundamental frequency is known.

The FLC adapts to the amplitude and phase of an oscillation in the primary input and tracks their changes. It is computationally inexpensive, inherently zero-phase, and has an infinite null. For M=1, the algorithm can be viewed as an adaptive notch filter, the width of the notch created at ω_0 being directly proportional to μ [15]. However, cancellation of periodic interference with the FLC depends on determination of the proper reference frequency, ω_0. The FLC cannot estimate the proper ω_0 value on-line because the tremor frequency is not known a priori. Making the FLC useful for tremor canceling during human-machine control requires a method to adapt the reference frequency to the primary input frequency.

4.3 WFLC

Due to the non-stationary nature of tremor, effective canceling requires adapting to change in both frequency and amplitude, whereas the FLC operates at a preset fixed frequency. To provide the needed versatility, the FLC has been extended to the case of time-varying frequency in the weighted-frequency Fourier linear combiner (WFLC), shown in Fig. 15. Riviere and Thakor modified FLC by replacing the fixed frequency, ω_0, of the FLC with another adaptive weight, ω_{0k}, that learns the input frequency via the LMS algorithm, much as the FLC weights learn the input amplitudes [15]. Like the FLC, the WFLC forms a dynamic truncated Fourier series model of the input. Unlike the FLC, the WFLC adapts the frequency of the model as well as its Fourier coefficients to match the input signal. The WFLC is, therefore, well suited to compensating for approximately periodic disturbances of unknown frequency and amplitude.

The WFLC algorithm is used to model the pathological tremor as a modulated sine wave to model the pathological tremor as a modulated sine wave with a time-varying frequency and amplitude. At every time step, error is measured between the tremor signal and the reference sine wave; depending on the error, frequency and amplitude of the reference sine-wave are adjusted. The WFLC algorithm is as follows,

$$x_{r_k} = \begin{cases} r\sin\left(r\sum_{t=0}^{k} w_{0_t}\right), & 1 \le r \le M \\ \cos\left((r-M)\sum_{t=0}^{k} w_{0_t}\right), & M+1 \le r \le 2M \end{cases}$$

$$\varepsilon_k = s_k - W_k^T X_k$$

$$w_{0_{k+1}} = w_{0_k} + 2\mu_0 \varepsilon_k \sum_{r=1}^{M} r\left(w_{r_k} x_{M+r_k} - w_{M+r_k} x_{r_k}\right)$$

$$W_{k+1} = W_k + 2\mu_1 X_k \varepsilon_k\,, \tag{4-8}$$

where $W_k = \left[w_{1_k}, w_{2_k}, \cdots, w_{2M_k}\right]^T$, $\quad X_k = \left[x_{1_k}, x_{2_k}, \cdots, x_{2M_k}\right]^T$

When M=1, the WFLC acts as a notch filter. μ_0 and μ_1 are adaptive gain parameters. Input amplitude and phase are estimated by the adaptive weight vector W_k, as in the FLC, while w_{0_k} estimates input frequency. It can be shown that, for sufficiently small μ_0, w_{0_k} converges to the frequency of a sinusoidal input signal [15]. If $\mu_0 = 0$, then the WFLC reduces to the FLC.

Using the frequency information obtained from the WFLC reference vector X_k a second set $\hat{W}_k$ of amplitude modulation and zero-phase tremor canceling is performed, as follows:

$$\hat{\varepsilon}_k = s_k - \hat{W}_k^T X_k$$
$$\hat{W}_{k+1} = \hat{W}_k + 2\hat{\mu}\ X_k \hat{\varepsilon}_k \qquad (4\text{-}9)$$
where $\hat{W}_k = \left[\hat{w}_{1_k} \cdots \hat{w}_{2M_k}\right]^T$.

Thus Eq. 4-9 operates essentially as a FLC with a time-varying reference frequency. A bias weight with a separate adaptive gain μ_b is used in parallel with this FLC to minimize distortion of lower frequency components. This overall system cancels tremor using an adaptive zero phase notch filtering approach which tracks changes in tremor frequency, amplitude, and phase.

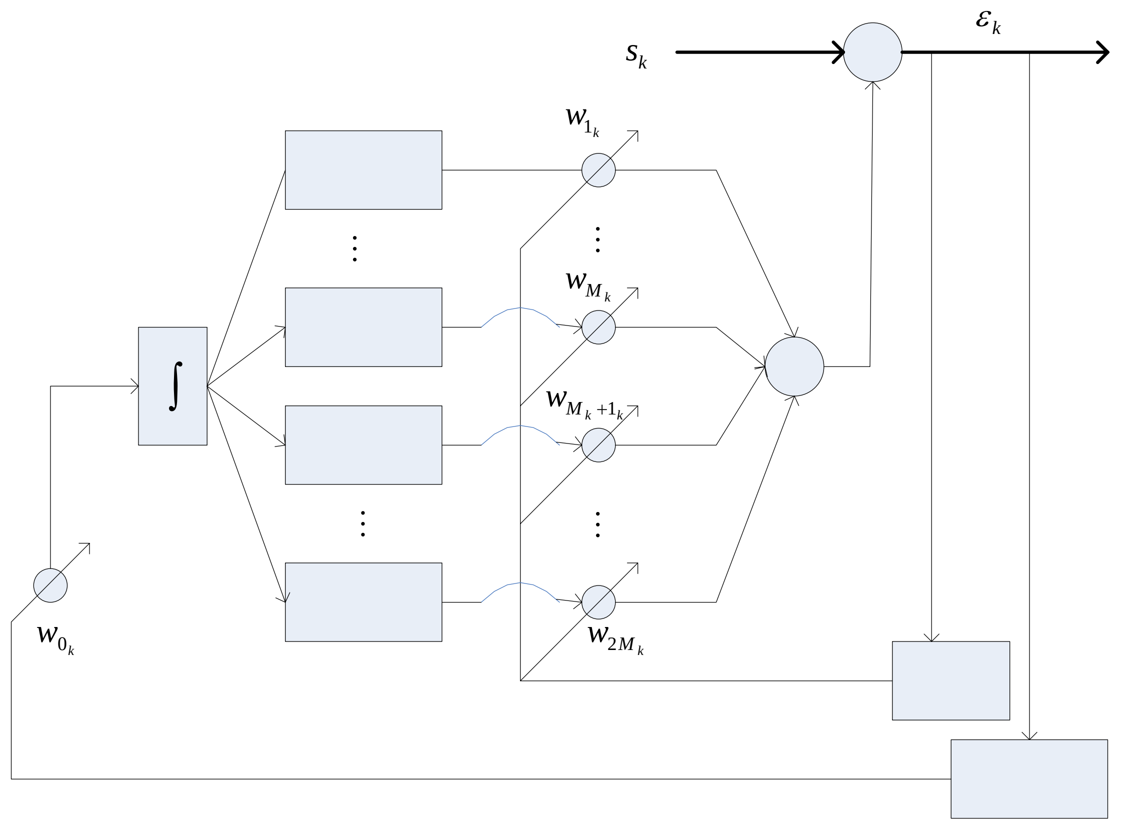


Figure 15: The Weighted-frequency Fourier Linear Combiner as an adaptive noise canceller.

The canceling FLC uses only the frequency information from the WFLC. The WFLC amplitude weight information is not used for the actual canceling. The band-pass pre-filter before the WFLC causes a small lag in the WFLC output, but this affects only the frequency, and not the amplitude, of the final canceling system. Since the WFLC frequency is constructed to change slowly for stability, and since tremor frequency, unlike tremor amplitude, tends to change only slowly, the effect of this delay on performance is negligible. The FLC operates with no pre-filtering, providing zero-phase tremor canceling.

4.4 MODIFIED WFLC

To improve CP tremor cancellation with WFLC techniques, we decided to use bandwidth information of CP tremor in the adaptive notch filter as well since we showed the importance of instantaneous bandwidth in the notch filter in Chapter 3. According to Riviere et al, when the value of adaptive gain μ increases, the bandwidth of amplitude weights from FLC also increases. To have a closed form expression of μ with respect to the bandwidth, the transfer function of FLC algorithm was needed. Fig. 16 shows the flow diagram of FLC system.

The transfer function of the path $y_k = 2\mu u(k-1)\cos(k\omega_0)$ from Fig.16 is

$$G(z) = 2\mu\left[\frac{z(z-\cos\omega_0)}{z^2 - 2z\cos\omega_0 + 1} - 1\right] = \frac{2\mu(z\cos\omega_0 - 1)}{z^2 - 2z\cos\omega_0 + 1}. \qquad \text{(4-10)}$$

Since we have a feedback loop from the joystick input to the noise canceller output, the feedback formula is

$$H(z) = \frac{1}{1+G(z)} = \frac{z^2 - 2z\cos\omega_0 + 1}{z^2 - 2(1-\mu)z\cos\omega_0 + 1 - 2\mu} \qquad (4\text{-}11)$$

Since Eq. 4-11 and Eq. 3-12 are the same, we can use the bandwidth equation to get the time-varying adaptive step parameter μ. We can set

$1 - 2\mu = 1.2 - 10*(0.02 + normalized\ \ IB)$ and then, $2\mu = 10*(0.02 + IB) - 0.2 = 10*IB$.

Therefore, $\mu = 5*normalized\ \ IB$

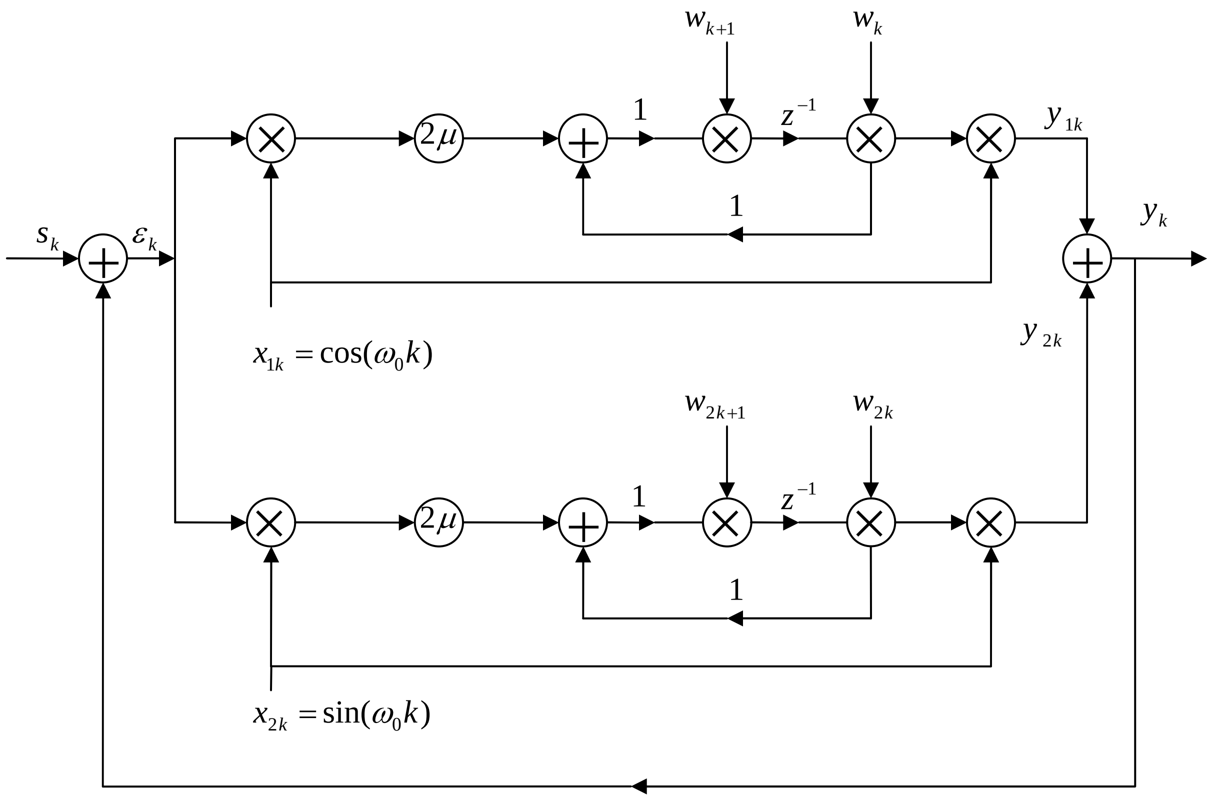


Figure 16: Flow diagram showing signal propagation in FLC.

Eq. 4-11 shows that the single frequency noise canceller has the properties of a notch filter at the reference frequency w_{0_k}. The zeros of the transfer function are located in the z-plane at $z = e^{\pm j\omega_0}$ and are precisely on the unit circle at angles of $\pm\,\omega_0\ rad$. The poles are located at

$$z = (1-\mu)\cos\omega_0 \pm j\left[(1-2\mu)-(1-\mu)^2\cos^2\omega_0\right]^{1/2} \qquad (4\text{-}12)$$

The poles are inside the unit circle at a radial distance $(1-2\mu)^{1/2}$, approximately equal to $1-\mu$, from the origin at the angle of

$$\pm\cos^{-1}\left[(1-\mu)-(1-2\mu)^{-1/2}\cos\omega_0\right]. \qquad (4\text{-}13)$$

Since the zeros lie on the unit circle, the depth of the notch in the transfer function is infinite at the frequency $\omega = \omega_0$. The sharpness of the notch is determined by a closeness of the poles to the zero. Corresponding poles and zeros are separated by a distance approximately equal to μ. Fig.17 shows the two different notch widths with two different μ values. The single frequency noise canceller is thus equivalent to a stable notch filter.

The overall CP suppression system is shown in Fig. 18. The FLC uses the frequency information of CP tremor from the WFLC and time-varying adaptive step gain from instantaneous bandwidth of CP tremor. The WFLC amplitude weight information is not used for the actual canceling. The band-pass pre-filter before the WFLC causes a small lag in the WFLC output, but this affects only the frequency, and not the amplitude, of the final canceling system. Since the WFLC frequency is constructed to change slowly for stability, and since tremor frequency, unlike tremor amplitude, tends to change only slowly, the effect of this delay on performance is negligible. The FLC operates with no pre-filtering, providing zero-phase tremor canceling.

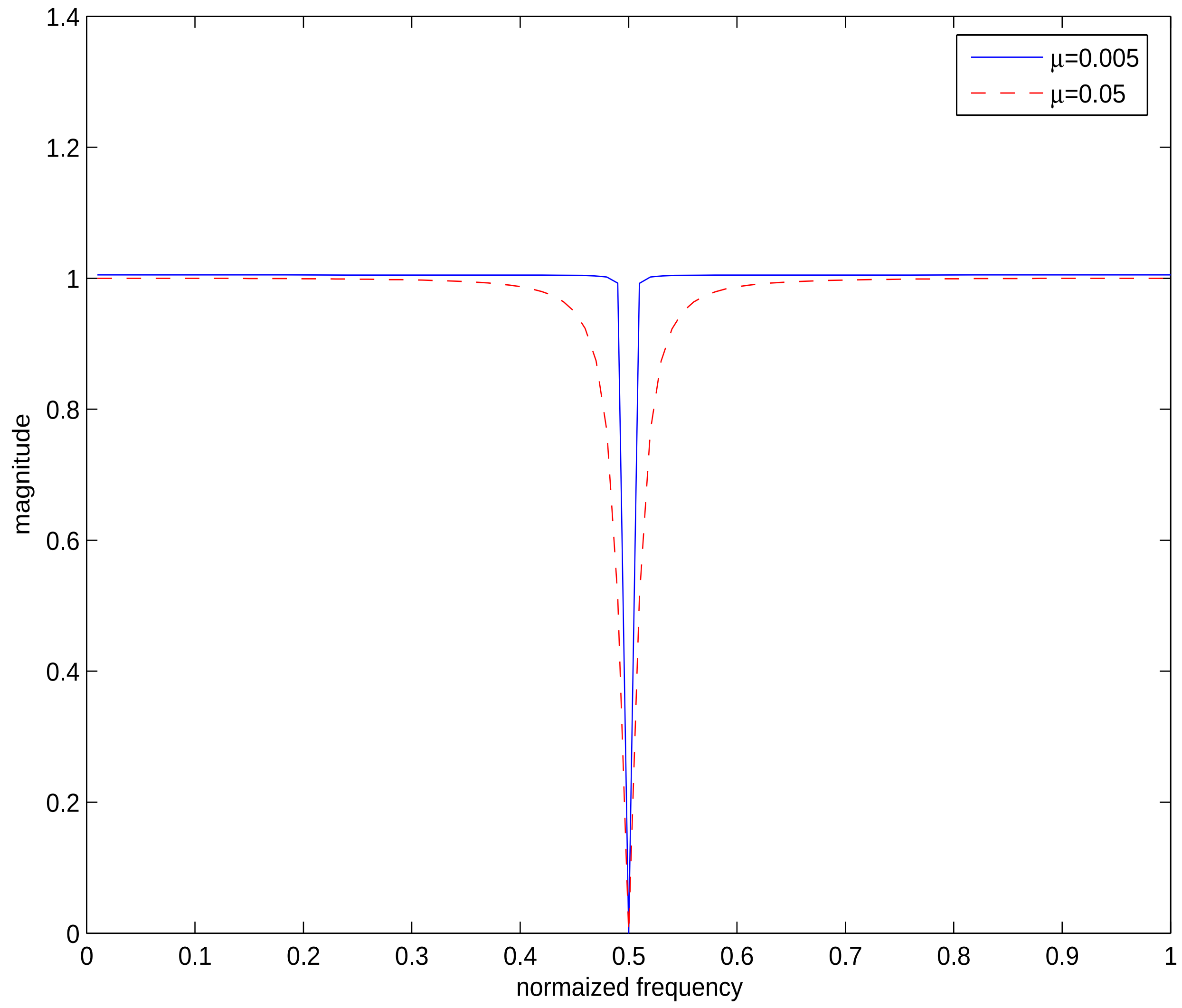


Figure 17: Magnitude of transfer function for the two different μ values at steady state condition.

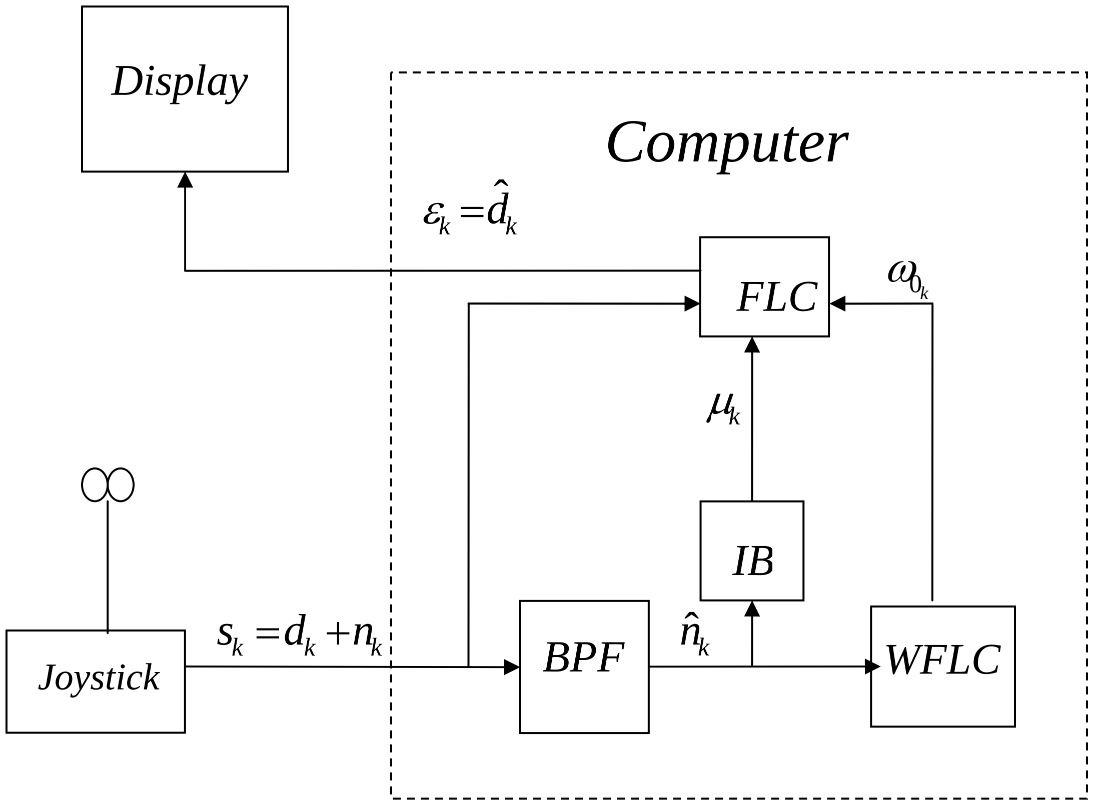


Figure 18: The block diagram of the experimental setup.

4.5 LOW-PASS FILTER

To compare the performance of an adaptive notch filter, we used a 3 Hz of low-pass filter in our virtual driving experiment. Most electric powered wheelchairs use a 3 Hz of low-pass filter to extract noises. The low-pass filter was also expected to suppress the CP tremor that normally has higher than 3 Hz frequency characteristics. We chose the third order of Butterworth low-pass filter because it is easy to implement and it has a good characteristics of low-pass filter. We used

100 Hz of a sampling frequency in our experiment. Fig. 19 and 20(a) show the magnitude and phase responses of the low-pass filter used in our experiment respectively. Phase delay depends on the target signal frequency. We calculated the time-delay of 1 Hz of intended signal from the phase response and the time-delay curve of the low-pass filter is shown in Fig. 20(b). From Fig. 21(b), we can read the time delay for the 1 Hz of intended signal as a 0.12 seconds. Therefore, we expected to have 0.12 seconds of time-delay when the low-pass filter was selected as a tremor suppression technique. The time-delay of 0.12 seconds created performance degradation on our virtual driving test.

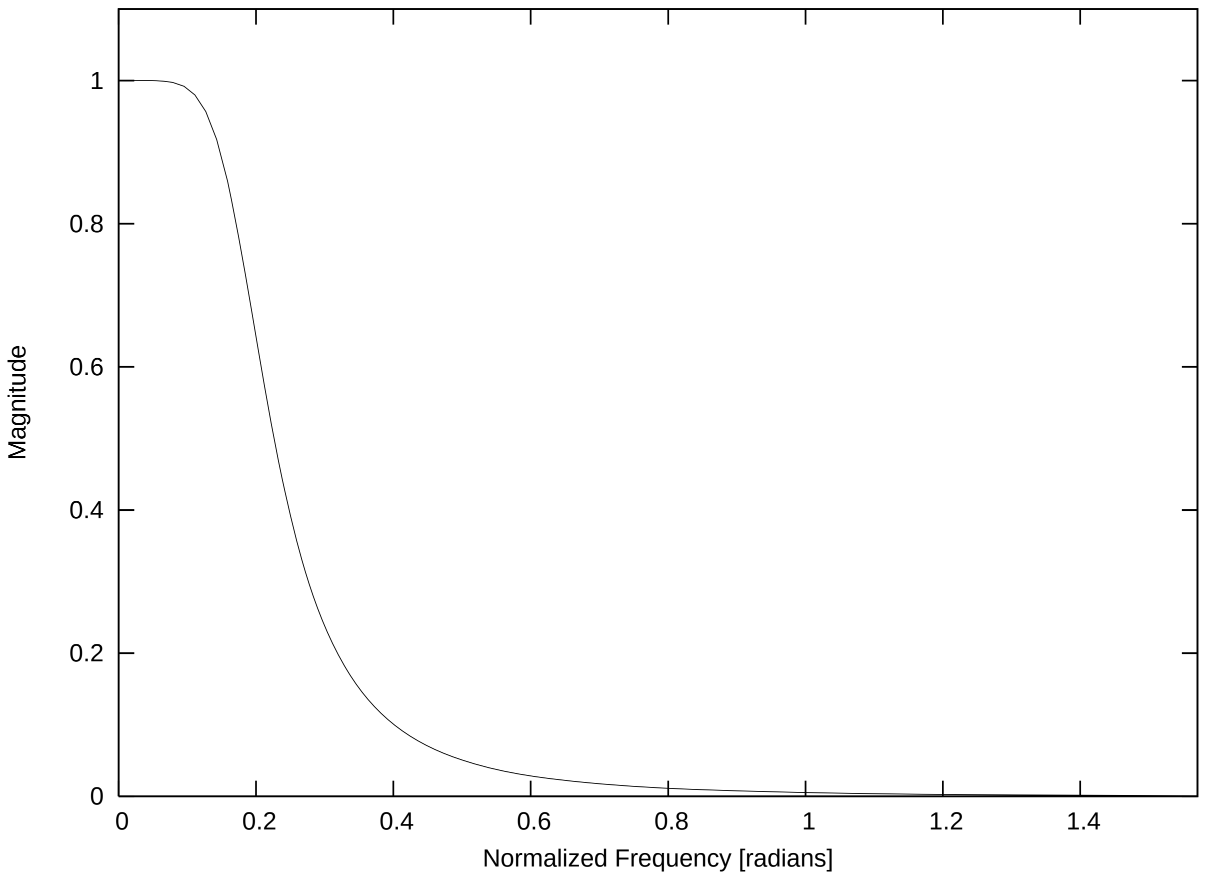
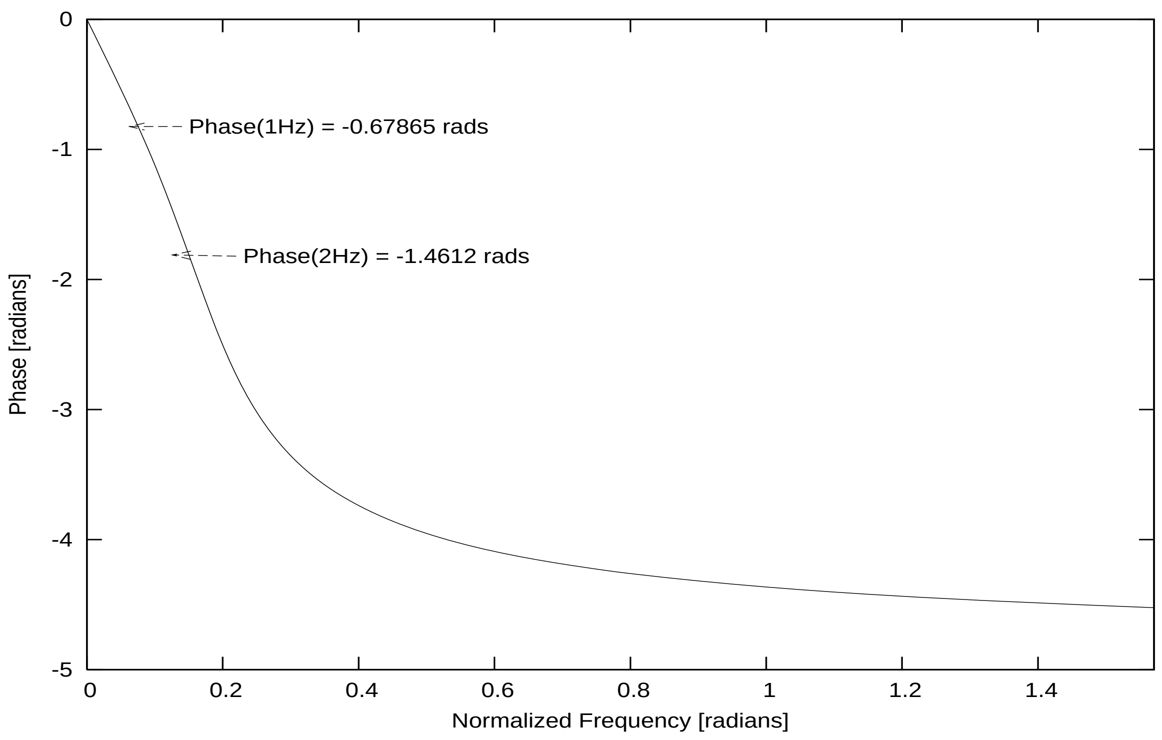
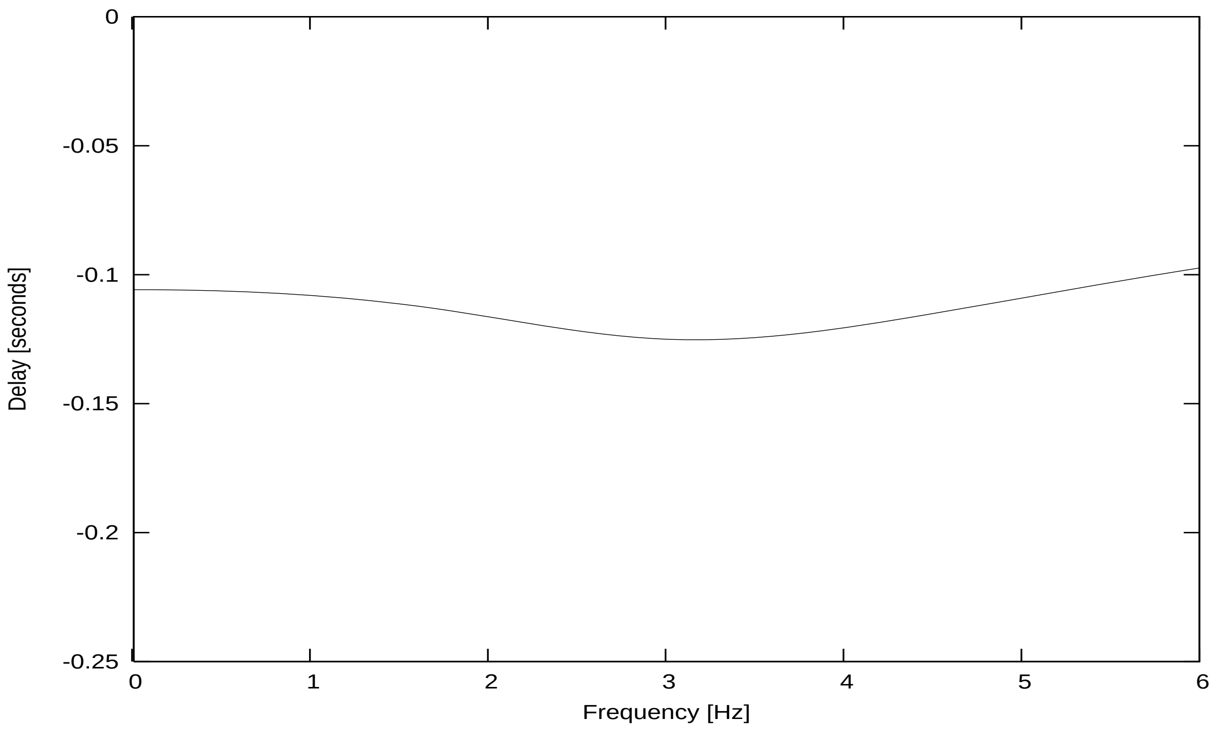


Figure 19: Magnitude response of 3 Hz Butterworth low-pass filter.



(a)



(b)

Figure 20: (a) phase response of 3 Hz low-pass filter and (b) its time-delay curve.

5.0 EXPERIMENT DEVELOPMENT

The aim of this research is to adapt a new filtering technique for electric powered wheelchair (EPW) users who have Cerebral Palsy (CP) disorder with an upper extremity tremor (including athetoid movement). The technique was expected to effectively suppress the unintended hand movement in the joystick signal of EPWs. We used an isometric joystick with a low-pass filter, weighted-frequency Fourier linear combiner (WFLC), and a modified WFLC to perform a virtual wheelchair driving test on a personal computer display in order to evaluate the efficacy of the new filtering techniques.

5.1 HUMAN SUBJECTS

Ten adult subjects with Cerebral Palsy (CP) pathological tremor were recruited and tested for this study. They were all EPW users with a standard position sensitive joystick. Eight of the subjects were men and two were women. 41 ± 7.2 years of the subjects was the mean $\pm$ standard deviation age and they had been using an EPW for 12 ± 6.9 years. Nine of the subjects were right handed while one was left handed. None of the subjects had used an isometric joystick before. Subjects were recruited via fliers distributed to United Cerebral Palsy (UCP) Pittsburgh. Testing was conducted in a quiet room at UCP Pittsburgh. All experimental procedures took place during one study visit that required approximately one hour per subject.

Subjects provided written informed consent via a consent form approved by the University of Pittsburgh's Institutional Review Board (IRB) for the Protection of Human Subjects. Some subjects who have a severe tremor requested to have a representative sign the consent form on their behalf. Provision was made in the consent form for a signature from the representative of an adult able to provide assent, but unable to provide a signature due to physical limitations. Subjects were notified of their right to request to discontinue the study, the interview, or refuse to answer questions at any time. Confidentiality was maintained through use of an ID number assigned to each subject and used on the data forms. Subjects' names and contact information were kept separate from the corresponding data forms.

5.2 EXPERIMENT DESIGN

The testing equipment for our study consisted of the following items:

- An adjustable height over-bed table;
- an isometric joystick;
- a notebook PC computer with a virtual driving software;
- a 19" flat panel display on an adjustable angle base.

Fig. 26 shows the test equipment. The table had an adjustable height feature with sufficient clearance to permit a close fit to the participant's wheelchair. The table surface allowed the joystick to be securely mounted at each subject's preferred position. The 19" flat panel display was mounted on the testing table. A tilt feature allowed the display to be angled for easy viewing. The positioning of the monitor stand relative to the subject and its tilt angle were optimized based on the subject's expressed preference for head position, comfort and visual acuity.

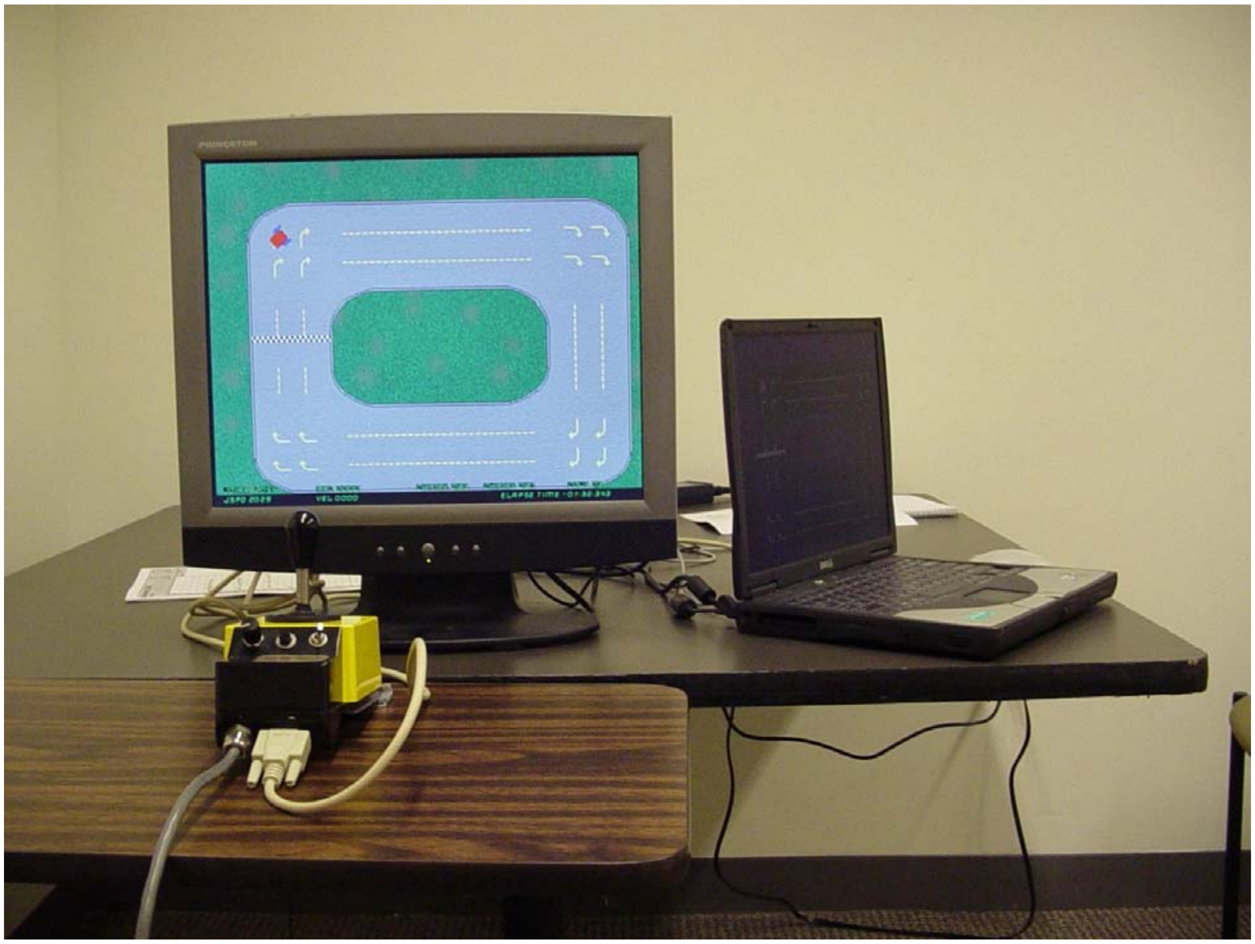


Figure 21: Experiment equipment setup.

5.2.1 Isometric joystick [28]

We used an isometric joystick that was developed at the Human Engineering Research Laboratories (HERL). The joystick produces a voltage output proportional to the force exerted on the stick by the user. The joystick voltage varies from 4.57 volts to 7.43 volts with 6 volts at the joystick's null position. A 12 bit counter was used to convert analog voltage to digital values. The decimal equivalent of a 12 bit binary number ranges from zero to $2^{12} - 1$, which equals 4095. The halfway point is 2047 and this was used to represent the neutral position on each axis. (From our data, we read the null position of joystick at 2047.) On the Speed axis (or x-axis), values greater than 2047 represented driving in the forward direction and values less than 2047 represented reverse driving. On the direction axis (y-axis), values above 2050 represented right turns and values below 2050 represented left turns. The isometric joystick was set to block any digital value higher than 3003 for the positive offset and 1092 for the negative to match with the voltage range of the standard position joystick for the Quickie P300 EPW. Underneath the operator controls, a DB-9 serial connector was interfaced to a notebook computer to acquire force data.

5.2.2 Software

We borrowed a virtual driving simulation software package from HERL and modified the software with Microsoft Visual Studio C++. Our main modification was the addition of three tremor suppression algorithms: (1) a low-pass filter, (2) WFLC, and (3) modified WFLC. We also added the capability of reading an input file, "vdd_wflc.ini", and writing output data files, "vdd_out_xxx.dat" and "vdd_out_xxx_ini.dat" for each trial. The "vdd_out_xxx.dat" and “vdd_out_xxx_ini.dat” output file names were updated with the number of the trial executed (for example “vdd_out_001.dat” and “vdd_out_001_ini.dat” for the first trial and so on). We had to generate the output file “vdd_out_xxx_ini.dat “ to trace what kind of setting we used for that particular trial. The main source code, “VirtualDriving_demo.cpp”, was compiled and created the executable file, “vdd_wflc.exe”. The executable file, “vdd_wflc.exe”, called the input ini file, ran the simulation, and created the output file. The main source codes are shown in Appendix A.1 and we highlighted and bolded the code that we added on the main source code. Additionally, we created six supplementary codes related to filtering and reading input settings. Those files are shown in Appendix A.2, A3, and A4. Our input file controlled the initial filter parameters and was shown in Fig. 27. An example of the output file is shown in Appendix B.1.

From the input file, we were able to select the type of suppression method to be used as well as the order of filter usage and could adjust the null position of the joystick for purposes of calibration. (Tremor suppression techniques were randomly selected by random generation with a Matlab script.) We also had three selections of prefiltering methods to extract noise: (1) no filter, (2) low-pass filter, and (3) median filter, although we always used a “no filter” option. We selected the commonly used third order of Butterworth filter with 3 Hz cutoff as a low-pass filter for comparison to our other filtering methods. We used the input file to adjust filter parameters for the WFLC algorithm after we observed the practice session. The output file showed all information about the simulation. We were able to see data index, time index, joystick x and y, low-pass filter x and y, WFLC x and y, modified WFLC x and y, chair x and y, crashes, instantaneous frequency, band-passed signal, instantaneous frequency, angular velocity, linear velocity, WFLC step gain, track error, and path number.

Figure 22: Example of an input file

```
# Filter parameters

tremor_removal_type=2           # 0(no filter), 1(lowpass), 2(wflc), 3(wflc modified)
filter_order=3                  # Filter order to be used (2 or 3)
joyStick_idle_pos.X=2047        # Joystick idle position X
joyStick_idle_pos.Y=2047        # Joystick idle position Y
sampling_freq(Hz)=100           # sampling frequency
prefilter_type=0                # 0(No prefiltering), 1(low-pass), 2(median)
prefilter.cut(Hz)=8             # cutoff frequency of low-pass prefilter
pre_median_order=3              # order of median prefilter (# of data samples)
intended_max_freq(Hz)=2.5       # intended movement max freq
tremor_freq_min(Hz)=2.5         # tremo minimum freq
tremor_freq_max(Hz)=8           # tremo maximum freq
alpha=0.01              # used by modified WFLC (removal_type 3) to control FLC's mu
#                       # to control mu of FLC: mu' = mu * (1 + alpha*BW),
#                       # BW = normalized tremor bandwidth at a moment (0 <= BW <= 1)

[WFLC]
# WFLC parameters (to track tremor frequency)
# Set mub to zero (or a very small value)
#
# mu     mub          m0          f0      M       w1      wM1
0.0003   0.      0.0000002     4      1       0.      0.

[FLC]
# FLC parameters (to track actual intended signal)
# 1. Set mu0 = 0 (or a very small value)
# 2. f0 will be from WFLC.
# mu value will be adjusted by tremor amplitude (envelope) by
#   mu' = mu * (1 + alpha * tremor_bandwidth)
# to control WFLC/FLC's the stop band width.
#
# mu     mub          m0          f0      M       w1      wM1
0.01     0.00002      0.          4       1       0.      0.
```

5.2.3 Virtual driving simulation

The virtual driving test was similar to a video game environment. The computer display showed a wheelchair sprite and a driving course. Subjects guided the modeled wheelchair with a joystick to follow the driving path. We used two virtual driving courses. One is a clockwise track course, shown in Fig. 28, and the other is right angle turning course, shown in Fig.29. The track course has four phases, starting from phase 1 up, phase 2 right, phase 3 down, and ending with phase 4 left. The right angle turn has two phases, starting with phase 1 up and ending with phase 2 right. Initially, we asked subjects to use a track course for their practice session. Some subjects had difficulty completing the track course. In these cases, we requested they use the right angle turning course instead. The courses were designed so that a proficient wheelchair driver could complete the course in approximately 1 minute without rushing.

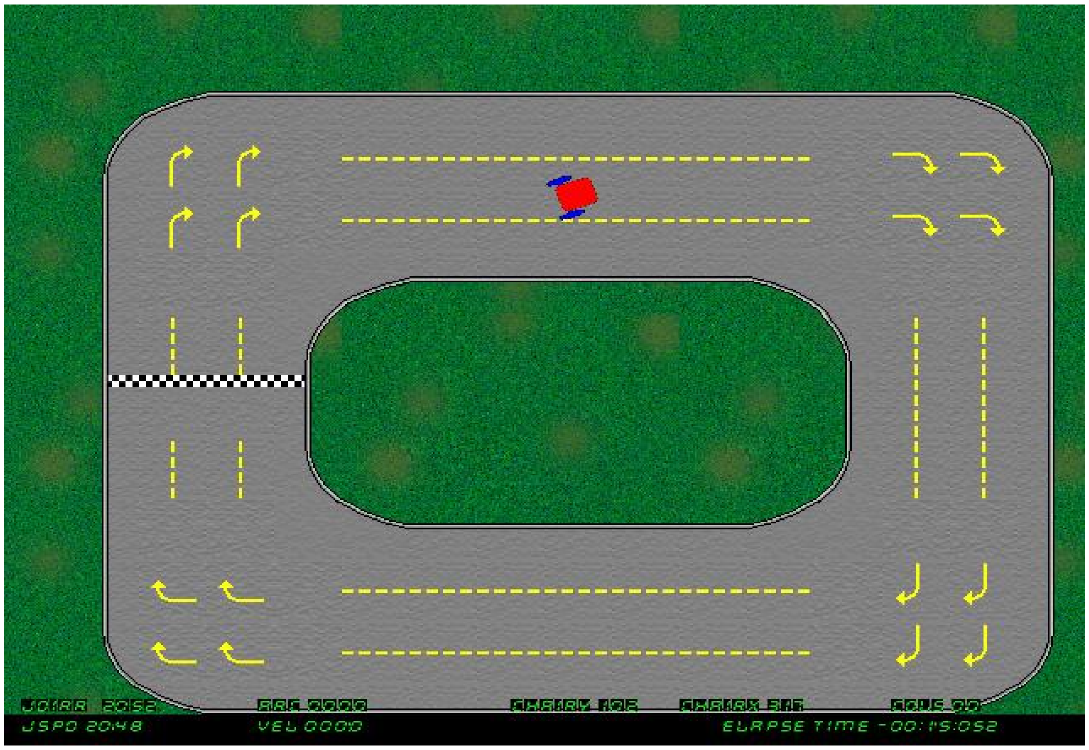


Figure 23: Track course and wheelchair sprite.

Since we did not have any information about the sizes of the virtual driving courses and the wheelchair sprite, we had to find numerical information on the virtual driving courses by using Adobe Photoshop. The resolution of the entire virtual screen is 640×480 and the origin (0,0) of the screen is located at the top left corner. The width of the path is 120 pixels and the width of the guided yellow lines is 40 pixels. For the track course, the center of the first phase (the straight path going upward) is at an x-axis position of 100. The second phase is at a y-axis position of 200. The third phase is at an x-axis position of 580 and the last phase is at a y-axis position of 400. For the right angle turn, the first phase is at an x-axis position of 100 and the second is at a y-axis position of 200.

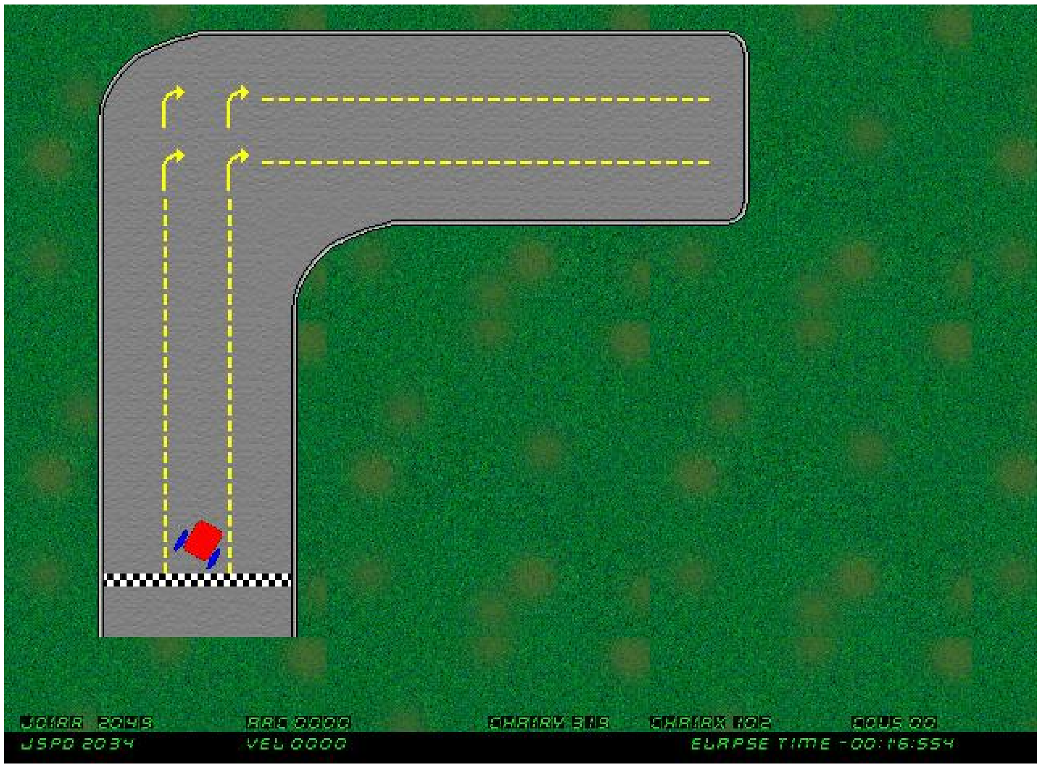


Figure 24: Right angle turn course and wheelchair sprite**.**

HERL used their speed profile data to model the Quickie P300 powered wheelchair manufactured by Sunrise Medical LLC. for the wheelchair sprite on the virtual driving simulation [28].

5.3 PROCEDURE

After the completion of the consent form, the practice testing session began by selecting the appropriate height of the table and the optimum location for the joystick. The computer monitor was mounted on an adjustable height table. A tilt feature allowed the monitor to be angled for easy viewing. The positioning of the monitor stand relative to the subject, the monitor height and the tilt angle were optimized based on the subject's expressed preference for head position, comfort and visual acuity. We demonstrated the movement of the testing joystick to the subject during the first practice session. All test subjects were evaluated by the researcher to determine their ability to operate the test joystick safely and properly. This procedure involved assuring that the individual could be comfortably seated to use the isometric joystick.

Test subjects were instructed on the proper use of the joystick and given up to 15 minutes to operate the device. We gave instructions on the proper use of the joystick during this familiarization period. In this period, we observed the raw joystick data and the outcome of WFLC algorithm with 5 different FLC μ values which are 0.001, 0.002, 0.005, 0.02, and 0.05 (these values guaranteed a stable system according to [30]) and drew a comparison graph by using Matlab software. After comparing the outcomes, we adjusted WFLC parameters to enable the subject to have the smallest tremor amplitude, as observed graphically among the 5 different μ values and maintained the parameters until the subject completed all trials. If after 15 minutes or less, the test subject demonstrated the ability to operate the device properly, the subject was given the opportunity to continue with the study. We instructed the subject driving the wheelchair sprite that remaining in the yellow guided lines was more important than completing the entire course quickly. If the subject had difficulty guiding the wheelchair sprite on the virtual track course, we switched the subject to the easier right angle turn course instead. Some subjects continued to have difficulty driving the right angle course. We tried to obtain as much data from each subject as possible despite these difficulties. Each subject completed 20 trials: 5 trials for each filtering method and 5 trials without any filter. After allowing for rest breaks between trials, subjects drove the test wheelchair through the course repeatedly until they completed the course 20 times.

5.4 DATA ANALYSIS

Mathworks Matlab7.0, Microsoft Excel, and SPSS were used to analyze the data. When evaluating performance on the virtual driving test using the three different filtering algorithms, the primary elements considered were accuracy and speed measurements according to [27], [28]. Accuracy was determined by the number of crashes into the walls on the course and mean square error (MSE) between a guided path and actual wheelchair trajectory. Speed was determined using the time needed to complete the course.

Given the small sample size, the normality of the distribution of the data would likely be difficult to verify statistically. Histograms were created to gain a general sense of the distribution of the data. Both parametric and non-parametric tests were performed to determine if any differences in results occurred between the two approaches. For parametric analysis, repeated-measures ANOVA was used to examine differences among the three filtering methods for each dependent variable (crashes, MSE, time to complete course) with an alpha level of 0.05. Post-hoc testing was performed using paired t-tests. The Bonferroni correction was used to compensate for multiple comparisons, therefore the t-test alpha level was set at 0.017. For non-parametric analysis, Friedman's test was used to examine differences among the three filtering methods for each dependent variable. Post-hoc test testing was achieved using the Wilcoxon Signed Ranks Test with a Bonferroni-corrected alpha level of 0.017.

6.0 RESULTS

In this chapter, we analyze the joystick data that we collected from the wheelchair users who have Cerebral Palsy (CP) disorder with pathological tremor in order to characterize the CP tremor. We also compared the outcomes of the virtual driving test to determine which of three tremor suppression techniques performed best: a low-pass filter, WFLC, or modified WFLC.

6.1 CHARACTERIZATION OF CP TREMOR

We used the joystick data from all ten subjects to determine the characteristics of CP tremor. Subjects #2, #7 and #8 had difficulty completing the track course because of their lack of experience on the isometric joystick control. They reported that the virtual driving test was uncomfortable for them, and were asked to try the right angle turn course instead. (A right angle turn note “RT” appears next to subjects #2, #7 and #8 on the result tables.). We summarized the characteristics of CP tremor we captured from the joystick data of ten subjects during the virtual test in Table 3. For the clinical evaluation of CP tremor, typically tremor power is used to identify abnormalities in CP tremor and classify the types of abnormalities observed. From Table 3, we observed that average CP instantaneous tremor frequency ranged from 4.05 Hz for subject #3 to 5.29 Hz for subject #9. We also measured average CP tremor power which is in a range between 12.13 for subject #2 and 111.9 for subject #1. By using CP tremor power, we were able to divide the ten subjects into two groups. The first group contains those subjects who had severe impairments. These were subjects #1, #3, #5, and #10. The second group contains those subjects who had mild impairments, including subjects #2, #4, #6, #7, #8, and #9. It matches with our observations during the experiment.

Table 3: Chracteristics of ten subjects and two CP groups.

Group Classification	Subject #	Average Tremor Frequency (IF), Hz	Average Conditional Standard Deviation of IF , Hz	Average Tremor Power (IP)
Severely Impaired	1	5.05	0.4150	111.90
	3	4.05	0.4293	86.82
	5	4.98	0.4204	51.71
	10	4.83	0.3615	40.83
Mildly Impaired	2 (RT)	5.25	0.1210	12.13
	4	4.77	0.1432	17.04
	6	4.29	0.1002	20.22
	7 (RT)	5.11	0.1460	15.32
	8 (RT)	5.02	0.1312	26.62
	9	5.29	0.1450	15.67

There was a bimodal distribution of the average conditional standard deviation of instantaneous CP tremor frequency data with the severely impaired group having a range of 0.3615 - 0.4293 and the mildly impaired group having a range of 0.1002 – 0.1460 (see Table 3). A Mann-Whitney U test was performed to determine if the two groups differed statistically from another. (Please note that for this and all subsequent analyses presented in Chapters 6 and 7, parametric analysis yielded the same results as non-parametric analysis, therefore only non-parametric test results are reported.) The severely impaired CP group showed higher average conditional standard deviation of instantaneous tremor frequency than the mildly impaired group with a p value of 0.01. From Table 3, we observed that the two groups also appear to differ in average tremor power, with the severely impaired group showing larger values than the mildly impaired group. Subject #6 has narrow range of instantaneous tremor frequency variation (0.1002 Hz) and subject #3 has large instantaneous tremor frequency variation (0.4293 Hz). Subjects #1, #3, #5, and #10 have higher average conditional standard deviation of instantaneous tremor frequency and average tremor power than subjects #2, #4, #6, #7, #8, and #9.

We also obtained joystick data from a spinal-cord injured (SCI) person with tremor and an unimpaired subject without tremor to show how hand movements differ among these subjects. Fig. 30 shows examples of different joystick signals of hand shakiness for an unimpaired subject in (a), a CP subject in (b), and a SCI subject in (c). While an unimpaired subject does not show any hand shakiness, a CP subject and a SCI subject have hand shakiness. A SCI subject has faster hand shakiness than a CP subject, but the amplitude variation is smaller as Aylor et al. concluded [12]. We presented this example to highlight the significance of the variability in joystick amplitudes and this example does not impact the interpretation of the experimental results.

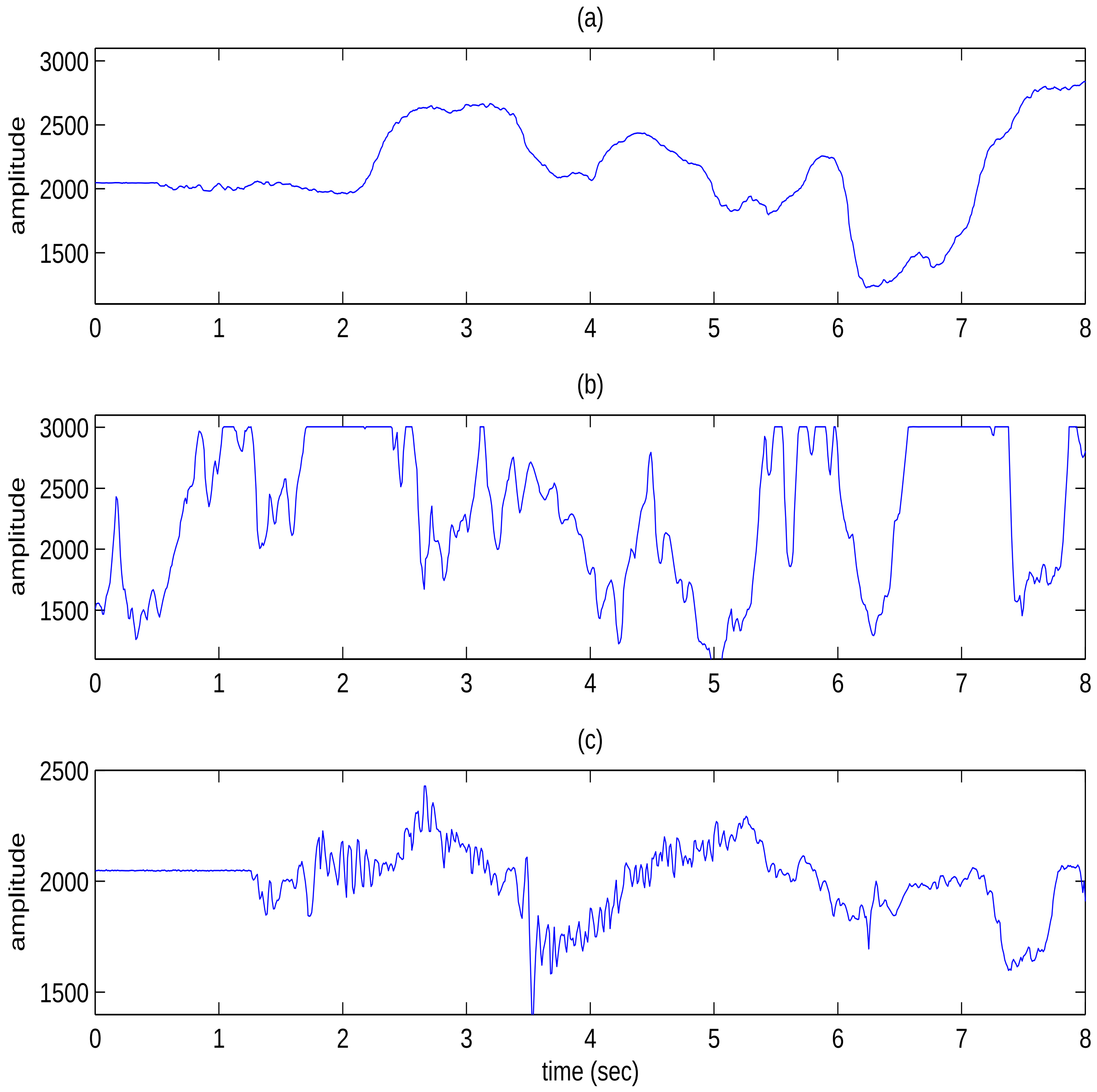


Figure 25: Different joystick signals among (a) unimpaired subject, (b) CP subject, and (c) SCI subject.

6.2 PERFORMANCE

Of the ten subjects, subject #2, #7, and #8 were unable to complete the track course because they were unable to coordinate the isometric joystick with the direction of the virtual wheelchair sprite, especially in phases 3 and 4. These subjects were asked to attempt the right angle turn course instead. The right angle turn course is easier than the track course because it has a shorter travel distance and requires the user to drive the wheelchair in fewer directions, making joystick coordination simpler. Subjects #7 and #8 still could not complete the right angle turn course 5 times for each filtering technique because they drove the virtual wheelchair extremely slowly. We were able to obtain only two sets of data for each filter from these subjects. Although we used two different courses, we are still able to compare the performance of each of the suppression methods because we are making comparisons within subjects. To measure performance on the virtual test, we used three elements including number of crashes to the wall, total traveled time to reach end line, and mean square error (MSE) to show deviation from the guided path.

Fig. 31 shows an example of one trial for subject #1. In Fig. 31, the red line is the actual wheelchair's trajectory shown with the yellow guided path and the green center line of the path. This example shows one crash on the third phase. The highest MSE of the wheelchair trajectory occurred in the fourth phase where the wheelchair was driven outside of the guided path most of time. Table 4 shows an example of summarized performance for subject #1. Subject #1 completed all 5 trials for each suppression method (no filter, low-pass filter, WFLC , and modified WFLC). Our output data file contains the performance information for this trial.

From our data analysis, male CP subjects drove the virtual wheelchair better than female CP subjects #2 and #8. Also, subjects who have experience with a video game performed better regardless of the extent of their impairments. Ironically, severely impaired CP subjects #1 and #3 drove the virtual wheelchair very quickly with few crashes.

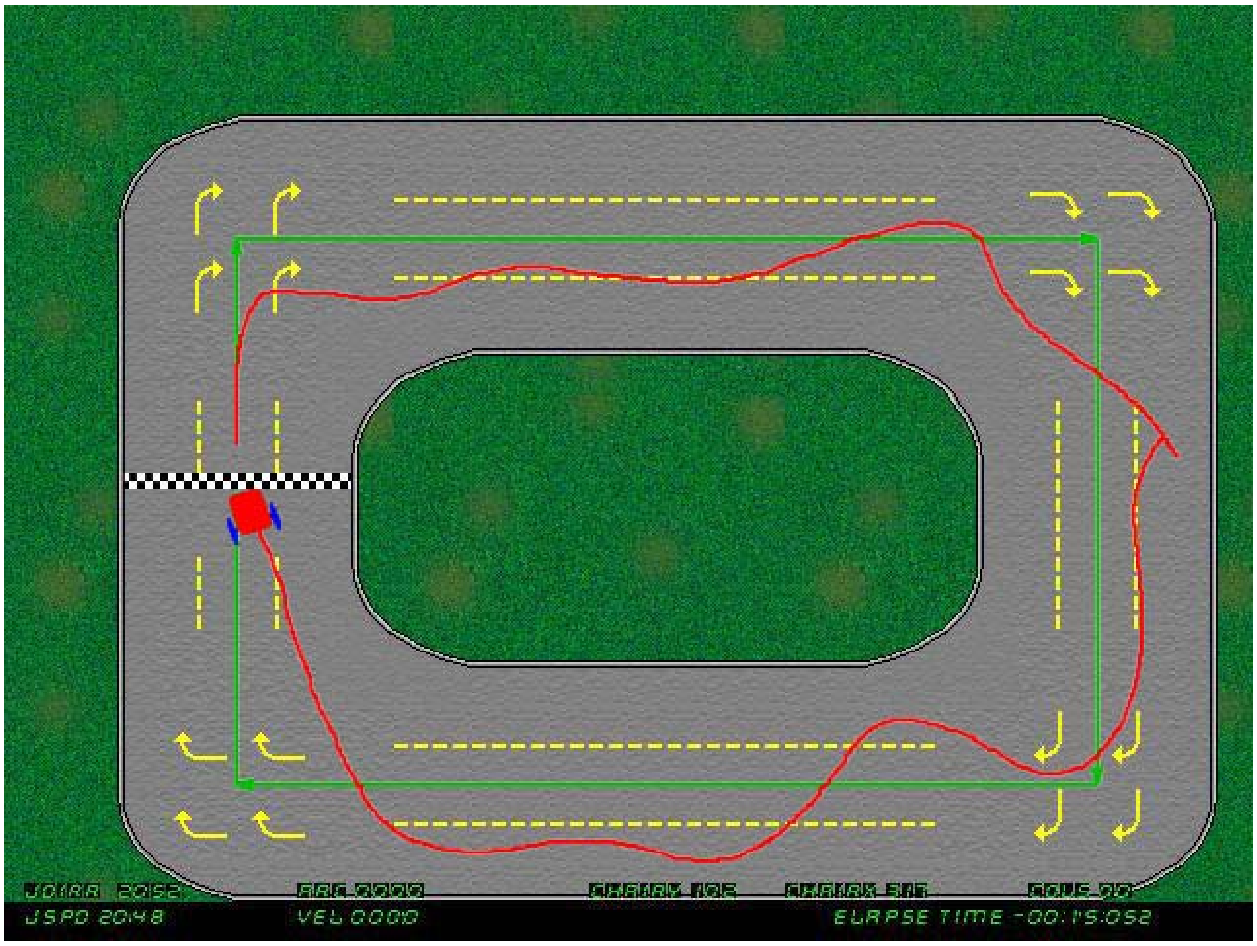


Figure 26: Example of third trial for subject #3 with WFLC algorithm.

Table 4: Summary of subject #1 output data

Trial	Total time, Sec.	# of Crashes	MSE	Average Tremor Freq. (IF), Hz	Average Tremor Power (IP)	Average Velocity	
						Linear	Angular
NoF1	19.79	0	0.16	5.13	124.11	94.73	13.89
NoF2	18.25	0	0.98	5.12	118.40	96.98	13.69
NoF3	18.15	0	1.98	4.64	108.92	94.72	11.59
NoF4	18.05	0	1.92	5.17	107.19	95.47	12.08
NoF5	17.40	0	2.31	5.00	118.21	101.44	13.32
Avg.	**18.32**	**0**	**1.47**	**5.01**	**115.37**	**96.67**	**12.58**
LPF1	21.77	0	0.80	5.07	114.04	86.68	15.16
LPF2	18.03	0	1.53	5.11	100.88	103.54	18.92
LPF3	23.74	1	5.21	5.23	105.89	76.07	17.82
LPF4	17.88	0	2.16	5.04	86.17	96.69	15.87
LPF5	19.24	0	3.02	5.20	117.51	93.50	15.99
Avg.	**20.13**	**0.2**	**2.54**	**5.13**	**104.90**	**91.30**	**16.75**
WFLC1	19.19	0	1.52	5.05	128.53	94.45	13.94
WFLC2	19.89	0	1.17	4.87	140.97	91.70	11.22
WFLC3	20.13	0	0.61	4.91	120.70	90.37	12.33
WFLC4	18.26	0	1.13	5.28	116.50	97.47	12.61
WFLC5	19.19	0	3.59	4.97	91.53	85.81	12.99
Avg.	**19.33**	**0**	**1.60**	**5.02**	**119.65**	**91.96**	**12.62**
Mod1	18.28	0	1.28	5.03	116.74	96.43	12.90
Mod2	18.54	0	0.50	5.00	102.92	95.54	14.59
Mod3	22.00	0	0.36	5.04	105.86	87.01	12.97
Mod4	19.06	0	1,24	5.09	126.39	93.31	12.98
Mod5	21.44	0	0.66	5.03	86.45	87.16	14.62
Avg.	**19.86**	**0**	**0.81**	**5.04**	**107.67**	**91.89**	**13.61**

6.2.1 Crashes

We counted the number of crashes into the wall during the virtual driving test. Table 5 shows the summary of the number of crashes for ten subjects. Friedman's test showed that a statistically significant difference existed among the three filtering methods (p=.006). Wilcoxon's Signed Ranks Tests were performed to identify differences between pairs of filtering methods. All subjects were able to complete the driving course with significantly fewer crashes using WFLC than the low-pass filter (p=.005). We did not observe any statistically significant performance difference between modified WFLC and the low-pass filter (p=0.022) and between WFLC and modified WFLC (p=0.373) since the p values were greater than the Bonferroni-corrected alpha level of 0.017.

Table 5: Average number of crashes for 20 trials and performance.

Subject #	LPF	WFLC	Mod. WFLC
1	0.2	0.0	0.0
2(RT)	4.8	1.8	1.6
3	10.4	2.0	1.2
4	4.4	4.2	4.0
5	17	7.6	5.8
6	15.4	8.0	9.2
7(RT)	9.0	3.5	4.0
8(RT)	12.5	4.5	6.5
9	8.0	7.3	8.4
10	4.0	3.3	4.2
Average	**8.6**	**4.2**	**4.5**

From Table 5, subjects #1, #2, #3, #5, #6, #7, and #8 showed fewer crashes with WFLC and modified WFLC than with the low-pass filter. In contrast, subjects #4, #9, and #10 had relatively similar performances for the three filtering techniques. The percent improvement in the average number of crashes for WFLC and modified WFLC with respect to the low-pass filter were 50.3% and 46.1% respectively. Subject #3 showed the best performance improvement for WFLC and modified WFLC except for that observed in subject #1. Although subject #1 had a severe tremor, he performed well in all filter conditions.

6.2.2 Elapsed time

We measured the total elapsed time for each trial and for each subject during the virtual driving test. Table 6 summarizes the traveled time for ten subjects. Friedman's test showed that a statistically significant difference existed among the three filtering methods (p=.002). Wilcoxon's Signed Ranks Tests were performed to identify differences between pairs of filtering methods. Subjects were able to complete the driving course faster using WFLC than the low-pass filter (p=.009). Similarly, subjects were faster with modified WFLC than the low-pass filter (p=.005). Again, we did not observe any significant performance difference between WFLC and modified WFLC (p=0.838).

From Table 6, all subjects except for subject #9 showed performance improvement with WFLC and modified WFLC, but subject #9 had better performance with modified WFLC then with the other two methods. Average improvements in traveled time for WFLC and modified WFLC with respect to the low-pass filter were 7.2 seconds and 6.4 seconds respectively. Subject #3 showed the best traveled time improvement for WFLC and modified WFLC except for that observed in subject #1.

Table 6: Average traveled time for 20 trials and performance.

Subject #	LPF	WFLC	Mod. WFLC
1	20.1	19.3	19.9
2(RT)	38.6	35.2	28.7
3	41.0	20.8	26.8
4	41.1	37.2	32.6
5	28.9	25.6	22.3
6	52.3	42.6	49.8
7(RT)	62.5	46.0	43.8
8(RT)	34.7	23.9	27.2
9	42.4	43.3	42.0
10	28.1	24.2	27.8
Average	**39.0**	**31.8**	**32.6**

6.2.3 Mean square error (MSE)

We measured the tracking error between a guided path and actual traveled wheelchair trajectory and calculated its mean square error (MSE) during the virtual driving test. Table 7 shows the summary of the MSE for ten subjects. Friedman's test showed that a significant difference existed among the three filtering methods (p=.001). Wilcoxon's Signed Ranks Tests were performed to identify differences between pairs of filtering methods. Subjects were able to follow the guided path better using WFLC than the low-pass filter (p=.007). Similarly, subjects were able to follow the guided path better with modified WFLC than the low-pass filter (p=.005). Once again, we did not observe any significant performance difference between WFLC and modified WFLC (p=0.022).

Table 7: Mean square error (MSE) between guided path and actual traveled path and performance.

Subject #	LPF	WFLC	Mod. WFLC
1	2.54	1.60	0.81
2(RT)	4.05	3.89	3.45
3	9.88	3.92	3.52
4	7.38	6.95	6.25
5	8.36	7.78	7.99
6	7.18	6.44	6.03
7(RT)	5.44	4.52	4.11
8(RT)	9.33	6.83	5.98
9	6.67	5.89	6.25
10	6.19	6.21	6.15
Average	**6.70**	**5.40**	**5.15**

From Table 7, all subjects except for subject #10 showed performance improvement with WFLC and modified WFLC. Filtering techniques did not make any difference in the performance of subject #10. Average performance improvements for WFLC and modified WFLC with respect to the low-pass filter were 17.9% and 25.0%, respectively. Subject #3 shows the best performance improvement again for WFLC and modified WFLC.

7.0 SUMMARY, DISCUSSION AND FUTURE RESEARCH

7.1 SUMMARY

We have introduced three filtering techniques on Cerebral Palsy (CP) tremor suppression while using an electric powered wheelchair joystick. One is a time-varying notch filter based on time-frequency analysis techniques for off-line tremor suppression, and the other is an adaptive notch filter based on the Fourier Linear Combiner (FLC) technique for on-line tremor suppression. We recruited ten subjects who have CP tremor and tested them in a virtual wheelchair driving environment. We characterized and classified CP tremor using the standard deviation of instantaneous CP tremor frequency as well as currently used CP tremor frequency and CP amplitude. We also demonstrated the importance of instantaneous bandwidth information on the study of CP tremor.

7.1.1 Characteristics of CP tremor

Frequency and amplitude have been major elements used to characterize CP tremor until now. Many researchers have used power spectral density to find the range of the CP tremor frequency and CP tremor peak amplitude to characterize the tremor and the range of CP tremor frequency from our joystick study matches with their work [1], [2], [3], and [7]. Since tremor signals possess spectral characteristics that vary with time, the spectral density of the signal does not indicate when the frequencies occur. Therefore, the power spectral density is not sufficient to observe the behavior of the time-varying signal by itself. We investigated this problem with time-frequency analysis. We measured instantaneous frequency, conditional standard deviation of instantaneous frequency, and instantaneous power from the time-frequency distribution of CP tremor. The use of such parameters gave a more complete and accurate characterization of pathological tremor. Furthermore, we found that conditional standard deviation of instantaneous tremor frequency could be a useful indicator to differentiate classes of tremor severity because we observed from our data that the severely impaired CP group had three times greater conditional standard deviation of instantaneous frequency of CP tremor than the mildly impaired CP tremor group. The severely impaired CP tremor group had wider frequency variation than the mildly impaired CP tremor group, and the mildly impaired CP tremor group had a roughly constant tremor frequency since the frequency variation is only about 0.1 Hz.

7.1.2 Performance of virtual driving test

Although many researchers have only concentrated on finding pathological tremor frequency information for their filtering algorithms, we considered bandwidth information as an additional constraint of the filter design since tremor amplitude has a time-varying envelope and the bandwidth of tremor varies with respect to the envelope. We designed a filter that responds to time-varying bandwidth information as well as time-varying frequency information of CP tremor. We showed improvement of CP tremor suppression with additional instantaneous bandwidth information from our simulation by using a synthetic signal.

In this study, we introduced two filtering techniques for CP tremor suppression: (1) a time-varying notch filter with TFD technique for the off-line experiment, and (2) an adaptive notch filter based on Fourier Linear Combiner (FLC) technique for the on-line experiment. The time-varying notch filter method uses information gained from the time-frequency distribution, whereby a filter notch is tuned to the IF, IB, and IP of the tremor noise. The time-varying notch filter with TFD estimates a tremor frequency, tremor bandwidth, and tremor amplitude and uses this information to suppress an unwanted tremor noise. TFD method works very well in off-line implementation, but it is not ideal in on-line implementation because of its time-delay problem and computational complexity.

On the other hand, the adaptive notch algorithm is generally well suited to active tremor control due to its computational simplicity and predictive capability. Riviere's Weighted-frequency Fourier Linear Combiner (WFLC) and our modified WFLC were implemented as an adaptive notch filter in the real-time tremor suppression system. Riviere's WFLC acts like an adaptive notch filter that controls a notch frequency and a notch depth, but our modified WFLC may be used to control one more element, notch bandwidth. We implemented both algorithms in the virtual wheelchair driving test along with a commonly used low-pass filter and compared their performances.

The most popular filtering algorithm used in the joystick of power wheelchairs is a low-pass filter with a 3 Hz cut-off frequency. However, a serious drawback of using a low-pass filter is the time-delay problem. We compared the outcomes of the virtual driving test to determine which of three tremor suppression techniques performed best: a low-pass filter, WFLC, or modified WFLC. We recruited and tested ten adult subjects with CP pathological tremor in this study. To measure performance on the virtual test, we used three elements including number of crashes to the wall, total traveled time to reach end line, and mean square error (MSE) to show deviation from the guided path. We observed poorer performance with the low-pass filter, and believe that this may be explained by the time-delay of 0.12 seconds from the low-pass filter. We believe the time-delay contributed to more crashes to the wall and longer total traveled time as well as a larger MSE between the actual traveled trajectory of the wheelchair sprite and the guided path. When the wheelchair sprite was approaching the wall, we believe the time-delay prevented the user from changing the direction of the wheelchair or stopping the wheelchair quickly. From our experiment, we showed that an adaptive notch filter, WFLC or modified WFLC, is a better choice than the low-pass filter for the suppression of CP tremor.

7.2 DISCUSSION AND FUTURE RESEARCH

7.2.1 Adaptive filters

The virtual test shows almost no performance difference between regular WFLC and modified WFLC although we have better simulation results with synthetic data. In the simulation we showed that the modified WFLC was performed better than the regular WFLC because the modified WFLC would suppress more tremor amplitude through its use of instantaneous tremor bandwidth information. However, subjects performed better with the regular WFLC for the number of crashes and total traveled time. These findings suggest that suppressing slightly more tremor amplitude may not make much performance difference in wheelchair driving applications. For the future research, we can apply our tremor suppression method to a handwriting device which has higher frequency intended movement than an electric powered wheelchair and where tremor amplitude influences handwriting output directly. We also can apply our system to spinal-cord injured people because they have 6-12 Hz of high tremor frequency. Typically, the adaptive noise-canceling algorithm works better for applications with high frequency ranges and pure sinusoidal signals. However if tremor frequency changes rapidly, WFLC algorithm will not response quickly enough to follow the frequency because of the limitation of step size μ in LMS technique and characteristics of an adaptive filter. In this case, an adaptive filter will track wrong frequency and filter out wrong signal. That is the disadvantage of using an adaptive filter over a low-pass filter.

Finally, we had to choose an adaptive gain μ of WFLC manually for all subjects because they each have their own tremor characteristics. Depending on the value of μ, the ability of tremor suppression differs. Although we chose the best performing μ value from the 5 trials of practice sessions, we cannot conclude that the adaptive gain value chosen would necessarily give us the best results. In future research, we have to consider updating μ value often and automatically.

7.2.2 Experimental setup

From our experiment, we found that our test settings were not ideal for our targeted subject population. All ten subjects regularly used an electric powered wheelchair for their daily mobility and used a conventional movement joystick. These subjects were presented with an isometric joystick control they had never seen or used before. In order to make subjects feel comfortable with an isometric joystick, we believe that subjects need to have used the joystick for a long perioder of time. However, it is difficult to leave the joystick with the subject for many hours for practical reasons. Therefore, using their current joystick, such as a conventional movement joystick, would be beneficial to perform the virtual driving test.

In addition, the current virtual driving course is not practical for subjects who have not used video games before because they have a hard time adjusting the direction of the testing wheelchair with their current joystick direction, especially when the testing wheelchair is in phases 3 and 4. This may explain why subject #1 and #3, who are experienced with video games, performed better than other subjects regardless of the filtering technique employed. Therefore, the best scenario would be to have the testing wheelchair face forward all the time and allow the testing course to rotate with respect to the direction of the wheelchair movement. Therefore, we need to modify our virtual driving simulation package to create more user-friendly software in future research.

There is another area we could improve to better simulate real wheelchair driving. We used a simply modeled wheelchair in our virtual driving experiment. The wheelchair was modeled with only a speed profile curve by HERL. That does not represent the whole behavior of an electric powered wheelchair. We can improve the virtual wheelchair modeling so that we may have more similar behavior to that of a real wheelchair.

APPENDIX A

SOURCE CODES

In this appendix, we included a virtual driving source code with supplementary codes. We highlighted and bolded the functions and the modifications that we made on the source code. We also added the codes and headers that related with the filter design and input files for our tremor suppression system.

A.1 MAIN SOURCE CODE (VIRTUALDRIVFING_DEMO.CPP)

```
//#include <iostream.h>
#include <time.h>
#include <direct.h>
#include <ddraw.h>
#include <math.h>
#include <sstream>
#include <dinput.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "cSurface.h"
#include "resource.h"
#include "cTrack.h"
#include "cHitChecker.h"
#include "cTextWriter.h"
#include "cKeyboard.h"
#include "stdafx.h"
#include "iostream"

/*---------------------------------------------------------------------------*/
#include "filter.h"
#include "filter_wflc.h"        // WFLC adaptive filter
#include "read_ini.h"           // Read parameters from ini file

FILE    *f_out; // result data collection
char    out_fname[128];
char    out_fpara[128];
```

```
int     out_count = 1;
unsigned  int Joy_X[4], Joy_Y[4];
extern t_INI_PARAMETERS ini_para;     // read_ini.c

void init_filters(void);
/*------------------------------------------------------------------------------*/

#define Pi 3.141592653589
#define ChairBOUND_POINTS 4
#define ChairSTATE_CRASHED_WALL 0
#define ChairSTATE_OK 1


const int MaxRight = 3000;
const int MaxLeft = 1100;
const int MaxForward = 3000;
const int MaxBackward = 1100;
const int ThresholdRight = 2100;
const int ThresholdLeft = 2000;
const int ThresholdForward = 2160;
const int ThresholdBack = 2000;


//DATA COLLECTION

//COMPUTER AND FILE MANGEMENT***************************************************************

FILE *fp;

int mkdir(const char* dirname);

//JOYSTICK VARIABLES *************
unsigned char InBuff_1[12];
unsigned int serial_sample_size_1 = 4;
unsigned int total_sample_counter_1 = 0;
//unsigned int sample_counter_1 = 0;
unsigned  int tempF1;
unsigned  int tempF2;

//Summary Data
char* Start_Time;
int Collision_Count_Sum;
int Complete_Time_Sum;

//Time-Series Data Buffers
//char* Data_OK[10];
long int Record_Counter = 0;
long Final_Record_Count = 0;

//ANIMATION VARIABLES
unsigned int ChairX,ChairY, InitX, InitY;
int CrashCount=0;
int displace = 0;
int arc = 0;
int position = 0;
int R_position = 0;
int iXImage=0, iYImage=0,iRow=0, iCol=0, iCol_18=0, iCol_72;
bool forward = true;
static bool startup = true;

//Karl's Ballistic modeling Variables
const double delta_t = 0.016;  //this will be samples per second
const double kp = 0.0015;
const double ki = 0.04;
const double km1 = 0.00325;
const double km2 = 1;

static double Vel_Err_Old = 0; //used in get position
static double Actual_Velocity_Old = 0;//v(i-1)
static unsigned int Actual_Position_Old = 0;//x(i-1)
```

```
//Serial port settings
HANDLE                  hPort_1;
HANDLE                  hPort_2;

DWORD                   Don_CommMask_Old_1;
DWORD                   Don_CommMask_Old_2;

COMMTIMEOUTS            Don_ctmOld_1;
COMMTIMEOUTS            Don_ctmOld_2;

COMMTIMEOUTS            Don_ctmNew_1;
COMMTIMEOUTS            Don_ctmNew_2;

HWND                    g_hMainWnd;
HINSTANCE               g_hInst;

LPDIRECTDRAW7           g_pDD = NULL;         // DirectDraw object
LPDIRECTDRAWSURFACE7    g_pDDSFront = NULL;  // DirectDraw frontbuffer surface
LPDIRECTDRAWSURFACE7    g_pDDSBack = NULL;   // DirectDraw backbuffer surface


DDBLTFX                 ddbfx;
RECT                    rcDest, rcObj;
cTrack                  pRaceTrack;
cSurface                g_surfChair;
cSurface                g_surfChair2;
cSurface                g_trajectory;
cSurface                m_sprChairExplode_0;
cSurface                m_sprCollisionCount_0;
cSurface                m_surfTitle;
cSurface                m_surfHelmet;
cHitChecker             m_hcBoundingPoly; // holding out-of-track area
cHitChecker             m_hcRaceChair;    // holding chair (40*40 Rect)

int iCrashTime=0;
int RAngle[4][40]={
    {0, 10, 20, 30, 40, 50, 60, 70, 80},
    {90, 100, 110, 120, 130, 140, 150, 160, 170},
    {180, 190, 200, 210, 220, 230, 240, 250, 260},
    {270, 280, 290, 300, 310, 320, 330, 340, 350}
};
char buffer[255];
int terror;
float ChairAngle=0,SpeedAdj,HistorySpeed;
int forwardflag[3]={0, 0, 0};
bool first=1, flag=0;
long lStart, iLast=0;
int m_iChairState=ChairSTATE_OK;
int k=0, nk=0;
long int BackX[10], BackY[10], NX=0,NY=0;
cTextWriter m_txDigital, m_txStatus;
char pTrackNames[7][30]={"", "Track_leftturn.rxt","Track_rightturn.rxt",
    "Track_straight.rxt", "Track_3pointturn.rxt", "Track_docking.rxt", "ddtest.rxt"};

char buf[50];
HWND InitWindow(int iCmdShow);
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam);

/*---------------------------------------------------------------------------*/
//  Function Prototypes
/*---------------------------------------------------------------------------*/
void ProcessIdle();
void Read_Serial_1();
//void Read_Serial_2();
void Get_New_Pos(int,int);
int InitDirectDraw();
void CleanUp();
void initialize_serial_1();
//void initialize_serial_2();
void DoIdle();
void ShowStats();
void Move_Chair(int,bool);
void Rotate_Chair(int,bool);
void Send_Arc_Pulse(bool);
void Send_Move_Pulse(bool);
void New_Send_Move_Pulse(bool);
void DO_JSK();

bool Speed_Sim(int);

/*----------------------------------------------------------------------------*/
//  Wheelchair control data
/*----------------------------------------------------------------------------*/
unsigned int frac_speed[140] = {
468,936,1404,1872,2341,2809,3277,3745,4213,4681,
5149,5617,6085,6554,7022,7490,7958,8426,8894,9362,
9830,10299,10767,11235,11703,12171,12639,13107,13575,14043,
14512,14980,15448,15916,16384,16852,17320,17788,18256,18725,
19193,19661,20129,20597,21065,21533,22001,22469,22938,23406,
23874,24342,24810,25278,25746,26214,26683,27151,27619,28087,
28555,29023,29491,29959,30427,30896,31364,31832,32300,32768,
33236,33704,34172,34640,35109,35577,36045,36513,36981,37449,
37917,38385,38853,39322,39790,40258,40726,41194,41662,42130,
42598,43067,43535,44003,44471,44939,45407,45875,46343,46811,
47280,47748,48216,48684,49152,49620,50088,50556,51024,51493,
51961,52429,52897,53365,53833,54301,54769,55237,55706,56174,
56642,57110,57578,58046,58514,58982,59451,59919,60387,60855,
61323,61791,62259,62727,63195,63664,64132,64600,65068,65536};

unsigned int Ratio_XY[18][2]={
{0,65536},
{5712,65287},
{11380,64540},
{16962,63303},
{22415,61584},
{27697,59396},
{32768,56756},
{37590,53684},
{42126,50203},
{46341,46341},
{50203,42126},
{53684,37590},
{56756,32768},
{59396,27697},
{61584,22415},
{63303,16962},
{64540,11380},
{65287,5712}};


int R_acel[140]={       //Radial acceleration
0,1,1,2,3,3,3,3,3,2,
3,2,3,2,2,2,2,2,2,1,
2,2,1,2,2,1,1,2,1,2,
1,1,1,2,1,1,1,1,1,1,
2,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,0,1,1,
1,1,1,1,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,2,
1,1,1,1,1,2,1,1,2,1,
1,2,1,2,1,2,2,1,2,2,
2,2,2,3,2,3,2,3,4,3,
4,4,5,6,6,8,10,13,19,37};

int R_acel_source[140]={
0,1,1,2,3,3,3,3,3,2,
3,2,3,2,2,2,2,2,2,1,
2,2,1,2,2,1,1,2,1,2,
1,1,1,2,1,1,1,1,1,1,
2,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,0,1,1,
1,1,1,1,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,2,
1,1,1,1,1,2,1,1,2,1,
1,2,1,2,1,2,2,1,2,2,
2,2,2,3,2,3,2,3,4,3,
4,4,5,6,6,8,10,13,19,37};


int acel[140]={  //Displacement acceleration
0,3,4,0,3,3,3,3,3,2,
3,2,3,2,2,2,2,2,2,1,
2,2,1,2,2,1,1,2,1,2,
1,1,1,2,1,1,1,1,1,1,
2,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,0,1,1,
1,1,1,1,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,2,
1,1,1,1,1,2,1,1,2,1,
1,2,1,2,1,2,2,1,2,2,
2,2,2,3,2,3,2,3,4,3,
4,4,5,6,6,8,10,13,19,37};

int acel_source[140]={
0,3,4,0,3,3,3,3,3,2,
3,2,3,2,2,2,2,2,2,1,
2,2,1,2,2,1,1,2,1,2,
1,1,1,2,1,1,1,1,1,1,
2,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,0,1,1,
1,1,1,1,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,2,
1,1,1,1,1,2,1,1,2,1,
1,2,1,2,1,2,2,1,2,2,
2,2,2,3,2,3,2,3,4,3,
4,4,5,6,6,8,10,13,19,37};

int Double_Res[72][2]={
{0,0},{0,0},{1,0},{1,0},{2,0},{2,0},{3,0},{3,0},{4,0},{4,0},{5,0},{5,0},{6,0},{6,0},{7,0},{7,0},{
8,0},{8,0},
{0,1},{0,1},{1,1},{1,1},{2,1},{2,1},{3,1},{3,1},{4,1},{4,1},{5,1},{5,1},{6,1},{6,1},{7,1},{7,1},{
8,1},{8,1},
{0,2},{0,2},{1,2},{1,2},{2,2},{2,2},{3,2},{3,2},{4,2},{4,2},{5,2},{5,2},{6,2},{6,2},{7,2},{7,2},{
8,2},{8,2},
{0,3},{0,3},{1,3},{1,3},{2,3},{2,3},{3,3},{3,3},{4,3},{4,3},{5,3},{5,3},{6,3},{6,3},{7,3},{7,3},{
8,3},{8,3}};

/*------------------------------------------------------------------------------*/
//  FILTERS
/*------------------------------------------------------------------------------*/
int tremor_removal_type;        // 0=no filtering, 1=wflc, 2=lowpass
int prefilter_type;             // low-pass prefiltering: 0=disable, 1=lowpass, 2=median
double prefilter_lpf_cutoff;    // low-pass prefilter cut-off freq.
int prefilter_median_order;     // median prefilter order

/*------------------------------------------------------------------------------*/
//  Filter Parameters
/*------------------------------------------------------------------------------*/

// Prefilter (LPF)
t_FILTER       s_pre_lowpass_filter[2];    // LPF prefilter with wide pass band
t_FILTER       s_median_filter[2];         // median filter parameters

//  WFLC Parameters
```

```
t_WFLC_PARA    s_wflc_para[2];       // WFLC parameters for 2-D input (X, Y)
t_WFLC_PARA    s_flc_para[2];        // FLC parameters for 2-D input (X, Y)
t_WFLC_PARA    s_flc2_para[2];       // modified FLC parameters for 2-D input (X, Y)
t_FILTER       s_bandpass_filter[2]; // bandpass filter parameters used in WFLC
t_FILTER       s_envelope_filter[2]; // median filter parameters

// LPF filters
t_FILTER       s_lowpass_filter[2];  // lowpass filter parameters for 2-D input

t_WFLC_PARA    *wflc[2];             // WFLC
t_WFLC_PARA    *flc[2];              // FLC
t_WFLC_PARA    *flc2[2];             // modified FLC
t_FILTER       *bandpass_f[2];       // for WFLC
t_FILTER       *envelope_f[2];
t_FILTER       *lowpass_f[2];        //
t_FILTER       *prelpf[2];           // lowpass prefilter with wide pass band
t_FILTER       *median_f[2];         // median prefilter


/*------------------------------------------------------------------------------*/
int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nCmdShow)
{
    int m_iOption;
    int m_iState=1;
    int iStart=0;
    int j;
    static long lLastOption = 0;
    cKeyboard m_Keyboard;

    g_hInst = hInstance;
    g_hMainWnd = InitWindow(nCmdShow);
    if(!g_hMainWnd)
       return -1;

    if(InitDirectDraw() < 0)
    {
       CleanUp();
       MessageBox(g_hMainWnd, "Could start DirectX engine in your computer. Make sure you have
at least version 7 of DirectX installed.", "Error", MB_OK | MB_ICONEXCLAMATION);
       return 0;
    }

    ReadIniFile ("vdd_filter.ini");   // read filter parameters

    ShowCursor(FALSE);
    m_Keyboard.Create(g_hInst,g_hMainWnd);

    pRaceTrack.Getbackbuffer(g_pDDSBack, g_pDD, g_hInst);
    g_surfChair.Create(g_pDD, 360, 160, RGB(0,0,0));
    g_surfChair.LoadBitmap(g_hInst, IDB_CHAIR_BLUE, 0, 0, 360, 160);
    g_surfChair2.Create(g_pDD, 360, 160, RGB(0,0,0));
    g_surfChair2.LoadBitmap(g_hInst, IDB_BITMAP2, 0, 0, 360, 160);
    m_txStatus.Create(g_pDD,g_hInst,3);

    DDBLTFX ddbfx;
    RECT    rcDest;

    ddbfx.dwSize = sizeof( ddbfx );
    ddbfx.dwFillColor = 0;
    SetRect(&rcDest, 0, 0, 640, 480);

    while( TRUE )
    {
       MSG msg;

       if( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
       {
           TranslateMessage( &msg );
           DispatchMessage( &msg );
       }
       else {
```

```
switch (m_iState) {
   case 1:
       g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL, &ddbfx);
       if(iStart == 0)
       {   // initialize
          m_iOption = 1;
          m_txDigital.Create(g_pDD,g_hInst,2);

          m_surfTitle.Create(g_pDD,350,190);
          m_surfTitle.LoadBitmap(g_hInst, IDB_TITLE);

          m_surfHelmet.Create(g_pDD, 28,26);
          m_surfHelmet.LoadBitmap(g_hInst, IDB_HELMET);

          iStart = 1;
       }
       m_surfTitle.Draw(g_pDDSBack, 145, 30);
       j=245;
       if(m_iOption <= 7)
       {
          m_txDigital.WriteText(g_pDDSBack, "LEFT TURN", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "RIGHT TURN", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "STRAIGHT FORWARD", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "STRAIGHT BACKWARD", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "DOCKING", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "TRACK", 200, j);
          j+= 26;
          m_txDigital.WriteText(g_pDDSBack, "QUIT", 200, j);
          j+= 26;
          //m_txDigital.WriteText(g_pDDSBack, "ENTER SETTINGS", 200, j);
       }
       m_Keyboard.Process();
       m_surfHelmet.Draw(g_pDDSBack, 163, 242 + ( ( (m_iOption%10) -1) * 26));
       if(GetTickCount() - lLastOption > 200)
       {
          if(m_Keyboard.CheckKey(DIK_DOWN))
          {
              m_iOption++;
              if(m_iOption > 7) m_iOption = 1;
              lLastOption = GetTickCount();
          }
          if(m_Keyboard.CheckKey(DIK_UP))
          {
              m_iOption--;
              if(m_iOption < 1) m_iOption = 7;
              lLastOption = GetTickCount();
          }
          if(m_Keyboard.CheckKey(DIK_RETURN) ||
              m_Keyboard.CheckKey(DIK_NUMPADENTER) ||
              m_Keyboard.CheckKey(DIK_SPACE))
          {
              lLastOption = GetTickCount();
              m_iState = 2;
              iStart = 0;
          }
       }
       if (m_iState==2)
          pRaceTrack.ReadFromFile(pTrackNames[m_iOption]);
       break;
   case 2:
       if(iStart == 0) {
          m_surfTitle.Destroy();
          m_surfHelmet.Destroy();

          initialize_serial_1(); //initialize serial port1 for Joystick
```

```
                    init_filters(); // initialize filters and open output file

                    iStart=1;
                    first=1;
                    pRaceTrack.GetStartPosition(&InitX,&InitY);
                    InitY -= 35;
                    ChairX=InitX;
                    ChairY=InitY;
                    iLast = GetTickCount();
                }
                m_Keyboard.Process();

                if(m_Keyboard.CheckKey(DIK_ESCAPE))
                {
                    m_iState = 1;
                    iStart = 0;
                    lLastOption = GetTickCount();
                    SetCommMask(hPort_1,Don_CommMask_Old_1); //clear serial port
                    //SetCommMask(hPort_2,Don_CommMask_Old_2); //clear serial port2
                    PurgeComm(hPort_1,PURGE_RXABORT);
                    //PurgeComm(hPort_2,PURGE_RXABORT);
                    SetCommTimeouts(hPort_1,&Don_ctmOld_1);
                    //SetCommTimeouts(hPort_2,&Don_ctmOld_2);
                    CloseHandle(hPort_1);
                    //CloseHandle(hPort_2);

                    fclose (f_out); // close the file opened in init_filters()

                    break;
                }
                switch(m_iOption)
                {
                    case 1:
                        if (!(ChairX<220)) {
                            DoIdle();
                        } else {
                            g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                            pRaceTrack.Draw(0, 0, -1, -1, 640, 460);
                            ShowStats();
                            g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                        }
                        break;
                    case 2:
                        if (!(ChairX>420))
                            DoIdle();
                        else {
                            g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                            pRaceTrack.Draw(0, 0, -1, -1, 640, 460);
                            ShowStats();
                            g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                        }
                        break;
                    case 3:
                        if (!(ChairY<60))
                            DoIdle();
                        else {
                            g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                            pRaceTrack.Draw(100, 100, -1, -1, 640, 460);
                            ShowStats();
                            g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                        }
                        break;
                    case 4:
                        if (!(ChairX>379))
                            DoIdle();
```

```
                          else {
                              g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                              pRaceTrack.Draw(0,0, -1, -1, 640, 460);
                              ShowStats();
                              g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                          }
                          break;
                      case 5:
                          if (!(ChairY<100))
                              DoIdle();
                          else {
                              g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                              pRaceTrack.Draw(0, 0, -1, -1, 640, 460);
                              ShowStats();
                              g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                          }
                          break;
                      case 6:
                          if (!(ChairX>60 && ChairX<180 && ChairY<250 && ChairY>240))
                              DoIdle();
                          else {
                              g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL,
&ddbfx);
                              pRaceTrack.Draw(0,0, -1, -1, 640, 460);
                              ShowStats();
                              g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40,
40);
                         }
                         break;
                      case 7:
                          PostQuitMessage(0);
                          exit(0);
                          break;
                  }
              case 8:
                  break;
              break;
           }
           ProcessIdle();
       }
    }

    CleanUp();

    return 0;
}

void initialize_serial_1()
{
    hPort_1 = CreateFile("COM1:",GENERIC_READ,0,0,OPEN_EXISTING,0,0);

// The DCB settings*****************
    DCB Port1DCB;

    GetCommState(hPort_1,&Port1DCB);
    Port1DCB.DCBlength = sizeof (DCB);

    Port1DCB.BaudRate=38400;
    Port1DCB.fBinary=TRUE;
    Port1DCB.fParity = FALSE;

    Port1DCB.fOutxCtsFlow = FALSE;
    Port1DCB.fOutxDsrFlow = FALSE;
    Port1DCB.fDtrControl = DTR_CONTROL_DISABLE;
    Port1DCB.fDsrSensitivity = FALSE;
    Port1DCB.fTXContinueOnXoff = TRUE;
    Port1DCB.fOutX = FALSE;
    Port1DCB.fInX = FALSE;

    Port1DCB.fErrorChar = FALSE;
    Port1DCB.fNull  = FALSE;
    Port1DCB.fRtsControl = RTS_CONTROL_DISABLE;

    Port1DCB.fAbortOnError = FALSE;
    Port1DCB.ByteSize = 8;
    Port1DCB.Parity = NOPARITY;
    Port1DCB.StopBits = ONESTOPBIT;

    SetCommState(hPort_1,&Port1DCB);

    //New CommMask setup
    SetCommMask(hPort_1,EV_RXCHAR);
    //Com Timing***********************************************

    //Save old
    GetCommTimeouts(hPort_1,&Don_ctmOld_1);

    //Install New timeout settings
  #if 0
    Don_ctmNew_1.ReadIntervalTimeout = 0.01;            //Max time between to characters
    Don_ctmNew_1.ReadTotalTimeoutMultiplier = .1;       //Max time for each byte
    Don_ctmNew_1.ReadTotalTimeoutConstant = 0;
  #else
    Don_ctmNew_1.ReadIntervalTimeout = 10;            //Max time between to characters
    Don_ctmNew_1.ReadTotalTimeoutMultiplier = 10;       //Max time for each byte
    Don_ctmNew_1.ReadTotalTimeoutConstant = 0;
  #endif

    SetCommTimeouts(hPort_1,&Don_ctmNew_1);
}

/*------------------------------------------------------------------------------*/
void init_filters(void)
{
    int joystick_idle_pos[2];
    double fc, fs;
    struct _stat stat_buf;
    char buf[256];
    FILE *fin;
    double fc1, fc2;
    int order;

    tremor_removal_type = (int)ini_para.tremor_removal_type.value;

    order = (int)ini_para.filter_order.value;

    joystick_idle_pos[0] = (int)ini_para.joystick_idle_pos_X.value;
    joystick_idle_pos[1] = (int)ini_para.joystick_idle_pos_Y.value;

    fs = ini_para.sampling_freq.value;
    fc = ini_para.intended_frea_max.value;

    // Prefilter
    prefilter_type = (int)ini_para.prefilter_lpf_cutoff.value;
    prefilter_lpf_cutoff = ini_para.prefilter_lpf_cutoff.value;
    prelpf[0] = &s_pre_lowpass_filter[0];       // LPF prefilter with wide pass band
    prelpf[1] = &s_pre_lowpass_filter[1];       // LPF prefilter with wide pass band
    init_lowpass_filter (prelpf[0], order, prefilter_lpf_cutoff, fs);
    init_lowpass_filter (prelpf[1], order, prefilter_lpf_cutoff, fs);
    //
    median_f[0] = &s_median_filter[0];
    median_f[1] = &s_median_filter[1];
    clear_filter (median_f[0]);
    clear_filter (median_f[1]);
    prefilter_median_order = (int)ini_para.prefilter_median_order.value;
    median_f[0]->A.order = prefilter_median_order;
    median_f[1]->A.order = prefilter_median_order;
    // Initialize WFLC filter & run-time parameters
    wflc[0] = &s_wflc_para[0];  // for X-coordinate
    wflc[1] = &s_wflc_para[1];  // for Y-coordinate
    init_wflc (wflc[0], 0);    // for X
    init_wflc (wflc[1], 0);    // for Y
    wflc[0]->offset = 0;    // zero offset for WFLC for noise frequecy tracking
    wflc[1]->offset = 0;

    // Initialize band-pass filter to be used in WFLC
    bandpass_f[0] = &s_bandpass_filter[0];
    bandpass_f[1] = &s_bandpass_filter[1];
    fc1 = ini_para.tremor_freq_min.value;
    fc2 = ini_para.tremor_freq_max.value;
    init_bandpass_filter (bandpass_f[0], 2, fc1, fc2, fs);      // bandpass filter order 2
    init_bandpass_filter (bandpass_f[1], 2, fc1, fc2, fs);

    // Initialize envelope filter for bandpassed tremor
    envelope_f[0] = &s_envelope_filter[0];
    envelope_f[1] = &s_envelope_filter[1];
    init_envelope_filter (envelope_f[0], ini_para.wflc[0].f0, fs);
    init_envelope_filter (envelope_f[1], ini_para.wflc[1].f0, fs);

    // Initialize FLC filter & run-time parameters
    flc[0] = &s_flc_para[0];    // for X-coordinate
    flc[1] = &s_flc_para[1];    // for Y-coordinate
    init_wflc (flc[0], 1);    // for X (FLC)
    init_wflc (flc[1], 1);    // for Y
    flc[0]->offset = ini_para.joystick_idle_pos_X.value;
    flc[1]->offset = ini_para.joystick_idle_pos_Y.value;
    // Initialize modified FLC filter & run-time parameters
    flc2[0] = &s_flc2_para[0];    // for X-coordinate
    flc2[1] = &s_flc2_para[1];    // for Y-coordinate
    init_wflc (flc2[0], 1);    // for X
    init_wflc (flc2[1], 1);    // for Y
    flc2[0]->offset = ini_para.joystick_idle_pos_X.value;
    flc2[1]->offset = ini_para.joystick_idle_pos_Y.value;

    // init low-pass filter
    lowpass_f[0] = &s_lowpass_filter[0];
    lowpass_f[1] = &s_lowpass_filter[1];
    init_lowpass_filter (lowpass_f[0], order, fc, fs);
    init_lowpass_filter (lowpass_f[1], order, fc, fs);

    // Prepare output data file
    do {
       sprintf (out_fname, "vdd_out_%.3d.dat", out_count++);
       if (out_count > 1000) {
           out_count = 1;
           sprintf (out_fname, "vdd_out_%.3d.dat", out_count++);
           break;
       }
    } while (! _stat(out_fname, &stat_buf));    // Do not overwrite existing data files
    // Save parameter ini file contents used.
    sprintf (out_fpara, "vdd_out_%.3d.ini", out_count - 1);
    f_out = fopen(out_fpara, "w");
    fin = fopen ("vdd_filter.ini", "r");
    if (fin) {
       while (fgets(buf, 255, fin) != NULL) {
           fprintf (f_out, "%s", buf);
       }
       fclose(fin);
    }
    fclose(f_out);
    // Prepare output data column headings
    f_out = fopen(out_fname, "w");
    fprintf (f_out, "#   No    Time       X       Y");
    fprintf (f_out, "    Lowpass(X,Y)       WFLC(X,Y)   mod_WFLC(X,Y)"
               "  ChairX  ChairY Crashes   band(X) f0(tremor) ARC   VEL      mu'\n");
}
/*--------------------------------------------------------------------------------*/
```

```
void Read_Serial_1() //**********************************************************
{
    DWORD dwBytesRead_1;
    DWORD dwCommEvent_1;
    DWORD dwError_1;
    COMSTAT cs_1;
    static Last_Sample_ms;
    static counter;
    int i;
    double x[2], y[2];
    double env[2], t;

    if (WaitCommEvent(hPort_1,&dwCommEvent_1,NULL))//waits for any comm event
    {
       if (dwCommEvent_1 & EV_RXCHAR) // detects whether the EV_RXCHAR flag is set
       {
           ClearCommError(hPort_1,&dwError_1,&cs_1);
           ReadFile(hPort_1,InBuff_1,serial_sample_size_1,&dwBytesRead_1,NULL);
           ClearCommError(hPort_1,&dwError_1,&cs_1);

           if((InBuff_1[0] & 240)!=16) {
              serial_sample_size_1 = 5;
              return;
           } else {
              tempF1 = 256 * (15 & InBuff_1[0]) + InBuff_1[1];
              tempF2 = 256 * (15 & InBuff_1[2]) + InBuff_1[3];

              //Last_Sample_ms = GetTickCount();
              Last_Sample_ms = GetTickCount() - iLast;

              counter++;//this snippet of code bleeds off extra data in the buffer
              if(counter == 2)
              {
                  serial_sample_size_1 = 8;
                  counter = 0;
              }
              else
                  serial_sample_size_1 = 4;

              // Prefiltering
              if (prefilter_type == 1) {
                  // Prefiltering with low pass
                  tempF1 = (int)filter (prelpf[0], tempF1);
                  tempF2 = (int)filter (prelpf[1], tempF2);
              } else if (prefilter_type == 2) {
                  // median prefilter
                  tempF1 = (int)filter_median (median_f[0], tempF1);
                  tempF2 = (int)filter_median (median_f[1], tempF2);
              }
              // No filtering
              Joy_X[0] = (int)tempF1;
              Joy_Y[0] = (int)tempF2;

              // Lowpass filtering
              Joy_X[1] = (int)filter (lowpass_f[0], tempF1 - ini_para.joystick_idle_pos_X.value)
                             + ini_para.joystick_idle_pos_X.value;
              Joy_Y[1] = (int)filter (lowpass_f[1], tempF2 - ini_para.joystick_idle_pos_Y.value)
                             + ini_para.joystick_idle_pos_Y.value;

              // WFLC filtering
              x[0] = filter (bandpass_f[0], tempF1 - ini_para.joystick_idle_pos_X.value);
              x[1] = filter (bandpass_f[1], tempF2 - ini_para.joystick_idle_pos_X.value);
              y[0] = filter_wflc (wflc[0], x[0]);
              y[1] = filter_wflc (wflc[1], x[1]);

              // FLC
              // feed the WFLC-tracked frequency value into the 2nd FLC
              flc[0]->w0 = wflc[0]->w0;
              flc[1]->w0 = wflc[1]->w0;
              Joy_X[2] = (int)filter_wflc (flc[0], tempF1);       // FLC
```

```
                Joy_Y[2] = (int)filter_wflc (flc[1], tempF2);

                // Modified FLC
                // feed the WFLC-tracked frequency value into the 2nd FLC
                flc2[0]->w0 = wflc[0]->w0;    // modified FLC
                flc2[1]->w0 = wflc[1]->w0;
                // feed the tracked frequency value to envelope filter
                t = (wflc[0]->w0 * wflc[0]->fs / 2 / PI);
                if (t > ini_para.tremor_freq_max.value) t = ini_para.tremor_freq_max.value;
                if (t < ini_para.tremor_freq_min.value) t = ini_para.tremor_freq_min.value;
                envelope_f[0]->fc1 = t;
                t = (wflc[1]->w0 * wflc[1]->fs / 2 / PI);
                if (t > ini_para.tremor_freq_max.value) t = ini_para.tremor_freq_max.value;
                if (t < ini_para.tremor_freq_min.value) t = ini_para.tremor_freq_min.value;
                envelope_f[1]->fc1 = t;

                env[0] = filter_envelope (envelope_f[0], x[0]);
                env[1] = filter_envelope (envelope_f[1], x[1]);
                flc2[0]->mu = (1 + ini_para.alpha.value*env[0])*ini_para.wflc[1].mu;
                flc2[1]->mu = (1 + ini_para.alpha.value*env[1])*ini_para.wflc[1].mu;

                Joy_X[3] = (int)filter_wflc (flc2[0], tempF1);      // modified FLC
                Joy_Y[3] = (int)filter_wflc (flc2[1], tempF2);

                fprintf (f_out, "%6d %7d", Record_Counter++, Last_Sample_ms);
                for (i = 0; i < 4; i++) {
                    if (Joy_X[i] > MaxRight) Joy_X[i] = MaxRight;
                    if (Joy_X[i] < MaxLeft) Joy_X[i] = MaxLeft;
                    if (Joy_Y[i] > MaxForward) Joy_Y[i] = MaxForward;
                    if (Joy_Y[i] < MaxBackward) Joy_Y[i] = MaxBackward;
                    fprintf (f_out, " %7d %7d", Joy_X[i], Joy_Y[i]);
                }
                fprintf (f_out, " %7d %7d %7d %10.3f %7.3f %5d %5d %12g\n",
                    ChairX, ChairY, CrashCount, x[0], (flc[0]->w0)*(flc[0]->fs)/PI/2.,
                    arc, displace, flc2[0]->mu);

                // Compensate Joystick control data by a selected filter
                tempF1 = Joy_X[tremor_removal_type];
                tempF2 = Joy_Y[tremor_removal_type];

                Get_New_Pos(tempF1,tempF2);
            }
        }
    }
}//end read_serial_1 function************************************************************

void Secure_Data()
{
    int result = mkdir("c:\\SubjectFiles");
    result = chdir("c:\\SubjectFiles");
    Final_Record_Count = Record_Counter;
    fp=fopen("trial.txt","w");

    fclose(fp);
}

void ShowStats()    //********************************************
{
    sprintf(buffer, "CHAIRX %03i", ChairX);
    m_txStatus.WriteText(g_pDDSBack, buffer, 400, 450);

    sprintf(buffer,  "CHAIRY %03i", ChairY);
    m_txStatus.WriteText(g_pDDSBack, buffer, 300, 450);
    //lower page display of values

    sprintf(buffer, "JDIRR  %04i", tempF1);
    m_txStatus.WriteText(g_pDDSBack, buffer, 10,450);

    sprintf(buffer, "JSPD %04i", tempF2);
    m_txStatus.WriteText(g_pDDSBack, buffer, 10,463);
    sprintf(buffer, "ARC %04i",arc);
    m_txStatus.WriteText(g_pDDSBack, buffer, 150,450);

    sprintf(buffer, "VEL %04i",displace);
    m_txStatus.WriteText(g_pDDSBack, buffer, 150,463);


    sprintf(buffer,  "COLS %02u", CrashCount);
    m_txStatus.WriteText(g_pDDSBack, buffer, 525, 450);

    m_txStatus.WriteText(g_pDDSBack, "ELAPSE TIME -", 425, 463);
    sprintf(buffer,"%02d:%02d:%003d", (lStart / 60000), (lStart / 1000) % 60, lStart % 1000);
    m_txStatus.WriteText(g_pDDSBack, buffer, 525, 463, true);
}

void DoIdle()// this function detects collisions
{
    g_pDDSBack->Blt(&rcDest, NULL, NULL, DDBLT_WAIT | DDBLT_COLORFILL, &ddbfx );
    pRaceTrack.Draw(0,0, -1, -1, 640, 460);
    lStart=GetTickCount()-iLast;
    ShowStats();
    rcObj.top  = ChairY; rcObj.bottom  = ChairY+40;
    rcObj.left = ChairX; rcObj.right   = ChairX+40;

    if(RectInRegion(pRaceTrack.m_hcRoadMap.hBoundingPoly, &rcObj) != 0)
    {
       if (iCrashTime == 0)
           iCrashTime = GetTickCount();
       // starts a clock to measure the crash inactivity period

       flag=1; //sets a flag indicating a crash inactivity is in progress
    }
    if (GetTickCount() - iCrashTime < 75 && flag==1)// here a crash is already started
    {
       New_Send_Move_Pulse(!forward);
       DO_JSK();
       g_surfChair2.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40, 40);
    } else {
       if(flag==1)
       {
           CrashCount++;
           MessageBeep(200);
           position =0;
       }
       flag=0;
       ChairX=InitX+ NX;
       ChairY=InitY+ NY;
       //ChairX=InitX+(int)NX;
       //ChairY=InitY+(int)NY;
       BackX[k]=NX;
       BackY[k]=NY;
       k++;
       if(k>9) k=0;

       DO_JSK();

       iCrashTime=0;
       g_surfChair.Draw(g_pDDSBack, ChairX, ChairY, iXImage, iYImage, 40, 40);
    }

}

void ProcessIdle()
{
    HRESULT hRet;

    static iLastBlit;

    if(GetTickCount() - iLastBlit < 20) return;
    while( 1 )
    {
       hRet = g_pDDSFront->Flip(NULL, 0 );
       if( hRet == DD_OK )
       {
          break;
       }
       if( hRet == DDERR_SURFACELOST )
       {
          g_pDDSFront->Restore();
       }
       if( hRet != DDERR_WASSTILLDRAWING )
       {
          break;
       }
    }
    iLastBlit = GetTickCount();
}

//******************************************************************************************
void Get_New_Pos(int direction,int speed)
{
    static bool Last_Dir = true;
    static bool Last_Rotation_Dir = true;
    bool clockwise = true;

    arc = 0;
    displace = 0;
    int Don_iRun = 0;
    int Don_iRise = 0;

    static int Speed_current = 0;
    static int Spin_current = 0;

    if(speed > ThresholdForward)
    {
       if (speed > MaxForward) speed = MaxForward;// check for over range
       Don_iRise = speed - ThresholdForward;
       forward = true;
       displace = Don_iRise/6;

       if(displace > 139) displace = 139;
    }

    //Check if moving backward
    if(speed < ThresholdBack)
    {
       if (speed < MaxBackward) speed = MaxBackward;// check for over range
       Don_iRise =  ThresholdBack - speed;
       forward = false;//forward is false if going backwards
       displace = Don_iRise/6;
       if(displace > 139) displace = 139;
    }

    //Check if turning Right
    if(direction > ThresholdRight)
    {
       if(direction > MaxRight) direction = MaxRight;
       Don_iRun = direction - ThresholdRight;
       clockwise = true;
       arc = Don_iRun/6;
       if (arc > 139) arc = 139;
    }

    //Check if turning Left
    if(direction < ThresholdLeft)
    {
       if (direction < MaxLeft) direction = MaxLeft;
       Don_iRun = ThresholdLeft - direction;
       clockwise = false;
       arc = Don_iRun/6;
```

```
        if(arc > 139) arc = 139;
    }

    Spin_current = arc;
    if((clockwise != Last_Rotation_Dir) && (R_position > 4))
    {
        Spin_current = 1;
        clockwise = !clockwise;
    }// trying to reverse spinning without fully stopping

    Last_Rotation_Dir = clockwise;

    if(Spin_current > R_position)//acceleration
    {
        if(R_acel[R_position] <= 0)
        {
            R_acel[R_position] = R_acel_source[R_position];//resets the pace variable
            R_position++; //no delay due to inertia
        }
        else
        {
            R_acel[R_position]--;//decrement the register
            if(R_acel[position]<=0)
            {
                R_acel[R_position] = R_acel_source[R_position];//resets the pace variable
                R_position++; //move to the next higher pace
                if(R_position>139) R_position = 139;
            }
        }
        arc = R_position;
    }

    if (Spin_current <= R_position)
    {
        R_position -= 3;
        if(R_position < 0) R_position = 0;
        arc = R_position;
    }

//*********************************************************************************************
    Speed_current = displace;

    if((forward != Last_Dir) && (position > 4))
    {
        Speed_current = 1;
        forward = !forward;
    }// trying to reverse direction without fully stopping

    Last_Dir = forward;

    if(Speed_current > position)//acceleration
    {
        if(acel[position] <= 0) {
            acel[position] = acel_source[position];//resets the pace variable
            position++; //no delay due to inertia
        } else {
            acel[position]--;//decrement the register
            if(acel[position]<=0) {
                acel[position] = acel_source[position];//resets the pace variable
                position++; //move to the next higher pace
                if(position>139) position = 139;
            }
        }
        displace = position;
    }

    if (Speed_current <= position)
    {
        position--;
        if(position < 0) position = 0;
        displace = position;
    }
    Rotate_Chair(arc, clockwise);
    Move_Chair(displace,forward);
}

//*****************************************************************************************
void Rotate_Chair(int ARC, bool clockwise1)
{
    static unsigned int Arc_heap = 0;

    Arc_heap += frac_speed[ARC];  //add the new distance on

    if  (Arc_heap  >(65536  +  (500  *  position)))  //  is  there  enough  acumulation  to  rotate  10 
degrees?
    {
       Arc_heap -=(65536+(500 * position));//subtract off the pulse
       Send_Arc_Pulse(clockwise1);// send the pulse
    }

}

//**************************************************************************
void Move_Chair(int DISPLACE, bool forward1)   //Determines whether to move the chair
{
    static unsigned int Displace_heap = 0;

    Displace_heap += frac_speed[DISPLACE];  //add the new distance on

    if(Displace_heap >65536) // is there enough acumulation to pulse a pixel?
    {
       Displace_heap -= 65536;//subtract off the pulse
       New_Send_Move_Pulse(forward1);// send the pulse
    }
}

//*******************************************************************************
//rotates the chair by changing the image used left or right 10 degs
void Send_Arc_Pulse(bool CLOCKWISE)
{
    if(CLOCKWISE)
    {
       iCol_72++;
       if(iCol_72 > 71) iCol_72 = 0;
    }

    if (!CLOCKWISE)
    {
       iCol_72--;
       if (iCol_72 < 0) iCol_72 = 71;
    }

    iCol=Double_Res[iCol_72][0];
    iRow=Double_Res[iCol_72][1];

    //rotates the chair by changing the image used
    iXImage=iCol*40;
    iYImage=iRow*40;
}

//*********************************************************************************************
void New_Send_Move_Pulse(bool FORWARD)
{
    int quadrant;
    static unsigned int X_heap;
    static unsigned int Y_heap;
    int X_Pulse = 0;
    int Y_Pulse = 0;

    iCol_18 = iCol_72%18;

    if(FORWARD)
```

```
        quadrant = iRow;
    else //This routine called if chair is driving backwards; flips angles 180 degrees
    {
        switch(iRow)
        {
            case 0:
                quadrant = 2;
                break;
            case 1:
                quadrant = 3;
                break;
            case 2:
                quadrant = 0;
                break;
            case 3:
                quadrant = 1;
                break;
        }
    }
    X_heap += Ratio_XY[iCol_18][0];
    Y_heap += Ratio_XY[iCol_18][1];

    if(X_heap >65536)
    {
        X_Pulse = 1; //set a pulse flag
        X_heap-=65536;//subtract off the pulse
    }

    if(Y_heap > 65536)
    {
        Y_Pulse = 1; //set a pulse flag
        Y_heap-=65536;//subtract off the pulse
    }

    switch(quadrant)
    {
        case  0:
            NX+= X_Pulse;
            NY-= Y_Pulse;     // advance coordinates
            break;
        case  1:
            NX += Y_Pulse;     // advance coordinates
            NY += X_Pulse;
            break;
        case  2:
            NX -= X_Pulse;     // advance coordinates
            NY += Y_Pulse;
            break;
        case  3:
            NX -= Y_Pulse;     // advance coordinates
            NY -= X_Pulse;
            break;
    }
}

HWND InitWindow(int iCmdShow)
{
    HWND      hWnd;
    WNDCLASS  wc;

    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = g_hInst;
    wc.hIcon = LoadIcon(g_hInst, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH )GetStockObject(BLACK_BRUSH);
    wc.lpszMenuName = TEXT("");
    wc.lpszClassName = TEXT("VirtualDriving_demo");
    RegisterClass(&wc);
    hWnd = CreateWindowEx(
       WS_EX_TOPMOST,
       TEXT("VirtualDriving_demo"),
       TEXT("Virtual Driving"),
       WS_POPUP,
       0,
       0,
       GetSystemMetrics(SM_CXSCREEN),
       GetSystemMetrics(SM_CYSCREEN),
       NULL,
       NULL,
       g_hInst,
       NULL
    );

    ShowWindow(hWnd, iCmdShow);
    UpdateWindow(hWnd);
    SetFocus(hWnd);

    return hWnd;
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
   switch (message)
   {
       case WM_KEYDOWN:
           if(wParam == VK_ESCAPE)
           {
              Secure_Data();
              PostQuitMessage(0);
              return 0;
           }
           break;
       case WM_DESTROY:
           PostQuitMessage(0);
           return 0;
   }

   return DefWindowProc(hWnd, message, wParam, lParam);
}

int InitDirectDraw()
{
    DDSURFACEDESC2 ddsd;
    DDSCAPS2       ddscaps;
    HRESULT        hRet;

    // Create the main DirectDraw object.
    hRet = DirectDrawCreateEx(NULL, (VOID**)&g_pDD, IID_IDirectDraw7, NULL);
    if( hRet != DD_OK )
       return -1;

    // Get exclusive mode.
    hRet = g_pDD->SetCooperativeLevel(g_hMainWnd, DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN);
    if( hRet != DD_OK )
       return -2;

    // Set the video mode to 640x480x16.
    //hRet = g_pDD->SetDisplayMode(640, 480, 16, 0, 0);
    hRet = g_pDD->SetDisplayMode(640, 480, 16, 0, 0);
    if( hRet != DD_OK )
       return -3;

    // Prepare to create the primary surface by initializing
    // the fields of a DDSURFACEDESC2 structure.
    ZeroMemory(&ddsd, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP | DDSCAPS_COMPLEX;
    ddsd.dwBackBufferCount = 1;

    // Create the primary surface.
    hRet = g_pDD->CreateSurface(&ddsd, &g_pDDSFront, NULL);
    if( hRet != DD_OK )
       return -1;

    // Get a pointer to the back buffer.
    ZeroMemory(&ddscaps, sizeof(ddscaps));
    ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
    hRet = g_pDDSFront->GetAttachedSurface(&ddscaps, &g_pDDSBack);
    if( hRet != DD_OK )
       return -1;

   return 0;
}

void CleanUp()
{
    g_surfChair.Destroy();

    if(g_pDDSBack)
       g_pDDSBack->Release();

    if(g_pDDSFront)
       g_pDDSFront->Release();

    if(g_pDD)
       g_pDD->Release();
}

void DO_JSK()// does read Serial 2 about 60 times per second
{
    static iLastRead1 = 0;
    if(GetTickCount() - iLastRead1 <10) return;
    Read_Serial_1();
    iLastRead1 = GetTickCount();
}
```

A.2 WFLC FILTER (FILTER_WFLC.C & FILTR_WFLC.H)

```
//*****************************************Begin filter_wflc.c **************************
#include <stdio.h>
#include <math.h>
#include <memory.h>

#include "filter.h"
#include "filter_wflc.h"
#include "read_ini.h"

extern t_INI_PARAMETERS ini_para;     // read_ini.c

/*---------------------------------------------------------------------------*/
int init_wflc (t_WFLC_PARA *wflc, int ini_id)
{
    int i;

#if 0
    memset ((char *)wflc, 0, sizeof (t_WFLC_PARA));  // clear filter memory
#else
    for (i=0; i<2*MAX_HARMONICS; i++)
    {
       wflc->w[i] = wflc->x[i] = 0;
    }
    wflc->wbias = 0;
    wflc->count = 0;
    wflc->error = 0;
    wflc->w0t = 0;
#endif

    wflc->M = ini_para.wflc[ini_id].M;
    if (wflc->M <= 0) return -1;  // data error
    wflc->fs = ini_para.sampling_freq.value;
    wflc->mu = ini_para.wflc[ini_id].mu;
    wflc->mub = ini_para.wflc[ini_id].mub;
    wflc->mu0 = ini_para.wflc[ini_id].mu0;
    wflc->w0 = 2.*PI*ini_para.wflc[ini_id].f0/ini_para.sampling_freq.value;
               // convert frequency f0 into radian unit w0
    wflc->w[1] = ini_para.wflc[ini_id].w1;   // weight vector
    wflc->w[wflc->M+1] = ini_para.wflc[ini_id].wM1;
    return 0;
}

// (Weighted) Fourier Linear Combiner Filter (support both WFLC and FLC)
/*--------------------------------------------------------------------------------*/
double filter_wflc (t_WFLC_PARA *wflc, double datum)
{
    double input, tfs;
    double sumcross;
    int i;
    double *w, *x;
    int M = wflc->M;

    w = wflc->w;
    x = wflc->x;

    input = datum - wflc->offset; /* remove offset */

    x[0] = 1;  // DC component (to adapt bias)
    wflc->w0t += wflc->w0;    // locate next sine & cosine samples
    for (i = 1; i <= M; i++)
    {
       x[i] = sin(i * wflc->w0t);
       x[M+i] = cos(i * wflc->w0t);
    }

    tfs = 0;   // tfs = truncated Fourier series (including DC component)
    for (i = 0; i <= 2*M; i++) tfs += w[i] * x[i];

    wflc->error = input - tfs;        // calculate error
    wflc->w[0] += 2 * wflc->mub * wflc->error;      // update bias weight (dc component)

    if (wflc->mu0 > 0) {
       /* update frequency weight, 'blind' to harmonics */
       sumcross = 0;
       for (i = 1; i <= M; i++) sumcross += i * (w[M+i]*x[i] - w[i]*x[M+i]);
       wflc->w0 -= 2 * wflc->mu0 * wflc->error * sumcross;
       //if (wflc->w0 <= 0) wflc->w0;
    }
    /* update amplitude weights */
    for (i = 1; i <= 2*M; i++)  w[i] += 2 * wflc->mu * wflc->error * x[i];

    wflc->count++;     // count data samples

    return (wflc->error + wflc->offset);
}

//***************************************End filter_wflc.c **************************

//*************************************Begin filter_wflc.h **************************

#ifndef _FILTER_WFLC_H_
#define _FILTER_WFLC_H_
```

```
#define MAX_HARMONICS  30

typedef struct
{
    // Fixed or initial parameters
    double mu;         // adaptive gain for amplitude
    double mub;               // adaptive gain for bias weight
    double mu0;               // adaptive gain for frequency
    int M;             // filter order (usually 1 or 2 for tremor removal)
    double fs;         // sampling frequency
    double fc1;               // lower bound cutoff frequency of noise signal (-1 not used)
    double fc2;               // upper bound cutoff frequency of noise signal (-1 not used)
    double offset;     // constant offset to be subtracted before processing
    // Parameters keep changing during run-time
    double w[2*MAX_HARMONICS];       // weights for terms of truncated Fourier series
    double x[2*MAX_HARMONICS];       // reference vector of truncated Fourier series
    double w0;         // reference frequency of Fourier series
    double w0t;               // ref. freq. * time elased
    double wbias;      // bias weight
    double error;      // error computed
    int    count;      // time count
} t_WFLC_PARA;

int init_wflc (t_WFLC_PARA *wflc, int ini_id);
double filter_wflc(t_WFLC_PARA *wflc, double datum);

#endif // _FILTER_WFLC_H_
```

```
 double w0t;           // ref. freq. * time elased
    double wbias;      // bias weight
    double error;      // error computed
    int    count;      // time count
} t_WFLC_PARA;

int init_wflc (t_WFLC_PARA *wflc, int ini_id);
double filter_wflc(t_WFLC_PARA *wflc, double datum);

#endif // _FILTER_WFLC_H_
```

```
//***************************************End filter_wflc.h **************************
```

A.3 IIR FILTERS (FILTER.C & FILTER.H)

```
//***************************************Begin filter.c **************************
#include <stdio.h>
#include <math.h>
#include <memory.h>

#include "filter.h"
#include "filter_wflc.h"
#include "read_ini.h"

extern t_INI_PARAMETERS ini_para;     // read_ini.c

/*------------------------------------------------------------------------------*/
int init_wflc (t_WFLC_PARA *wflc, int ini_id)
{
    int i;

#if 0
    memset ((char *)wflc, 0, sizeof (t_WFLC_PARA));  // clear filter memory
#else
    for (i=0; i<2*MAX_HARMONICS; i++)
    {
       wflc->w[i] = wflc->x[i] = 0;
    }
    wflc->wbias = 0;
    wflc->count = 0;
    wflc->error = 0;
    wflc->w0t = 0;
#endif

    wflc->M = ini_para.wflc[ini_id].M;
    if (wflc->M <= 0) return -1;  // data error
    wflc->fs = ini_para.sampling_freq.value;
    wflc->mu = ini_para.wflc[ini_id].mu;
    wflc->mub = ini_para.wflc[ini_id].mub;
    wflc->mu0 = ini_para.wflc[ini_id].mu0;
    wflc->w0 = 2.*PI*ini_para.wflc[ini_id].f0/ini_para.sampling_freq.value;
               // convert frequency f0 into radian unit w0
    wflc->w[1] = ini_para.wflc[ini_id].w1;   // weight vector
    wflc->w[wflc->M+1] = ini_para.wflc[ini_id].wM1;
    return 0;
}

// (Weighted) Fourier Linear Combiner Filter (support both WFLC and FLC)
/*------------------------------------------------------------------------------*/
double filter_wflc (t_WFLC_PARA *wflc, double datum)
{
    double input, tfs;
    double sumcross;
    int i;
    double *w, *x;
    int M = wflc->M;

    w = wflc->w;
    x = wflc->x;

    input = datum - wflc->offset; /* remove offset */

    x[0] = 1;  // DC component (to adapt bias)
    wflc->w0t += wflc->w0;    // locate next sine & cosine samples
    for (i = 1; i <= M; i++)
    {
       x[i] = sin(i * wflc->w0t);
       x[M+i] = cos(i * wflc->w0t);
    }

    tfs = 0;   // tfs = truncated Fourier series (including DC component)
    for (i = 0; i <= 2*M; i++) tfs += w[i] * x[i];

    wflc->error = input - tfs;        // calculate error
    wflc->w[0] += 2 * wflc->mub * wflc->error;      // update bias weight (dc component)

    if (wflc->mu0 > 0) {
       /* update frequency weight, 'blind' to harmonics */
       sumcross = 0;
       for (i = 1; i <= M; i++) sumcross += i * (w[M+i]*x[i] - w[i]*x[M+i]);
       wflc->w0 -= 2 * wflc->mu0 * wflc->error * sumcross;
       //if (wflc->w0 <= 0) wflc->w0;
    }
    /* update amplitude weights */
    for (i = 1; i <= 2*M; i++)  w[i] += 2 * wflc->mu * wflc->error * x[i];

    wflc->count++;     // count data samples

    return (wflc->error + wflc->offset);
}

//***************************************End filter.c **************************
```

```
//*************************************Begin filter.h **************************

#ifndef _FILTER_H_
#define _FILTER_H_

#define PI     3.141592653589

#define MAX_COEFFS     64
                      // maximum number of coefficients of filter

/*-----------------------------------------------------------------------------*/
// Polynomial
/*-----------------------------------------------------------------------------*/
typedef struct
{
    int order;                // order of polynomial
    double coeff[MAX_COEFFS]; // c[0] + c[1]*x + ... + c[N]*x^N
} t_POLYNOMIAL;

/*-----------------------------------------------------------------------------*/
// Filter Parameters
/*-----------------------------------------------------------------------------*/
typedef struct
{
    //   y[z] = H[z] * x[z]
    //
    //                ( a[0] + a[1]*z^-1 + a[2]*z^-2 + ... + a[N]*z^-N )
    // Filter:  H(z) = ----------------------------------------------------
    //                ( b[0] + b[1]*z^-1 + b[2]*z^-2 + ... + b[M]*z^-M )

    char name[32];
    // Filter coefficients
    t_POLYNOMIAL    A;        // numerator polynomial of H(z)
    t_POLYNOMIAL    B;        // denominator polynomial of H(z)

    // run-time parameters
    double x[MAX_COEFFS];     // past input x[n] (x[0] is the most recent)
    double y[MAX_COEFFS];     // past output y[n] (y[0] is the most recent)
    double fs;                // sampling frequency
    double fc1;                      // cut-off frequency (lower bound of pass band)
    double fc2;                      // cut-off frequency (upper bound of pass band)
       // low-pass:   0 = fc1 < fc2 <= fs/2  (cut-off at fc2)
       // high-pass:  0 < fc1 < fs/2 <= fc2  (cut-off at fc1)
       // band-pass:  0 < fc1 < fc2 < fs/2
       // band-stop:  0 < fc2 < fc1 < fs/2
       // notch:      0 < fc1 = fc2 < fs/2
} t_FILTER;

/*-----------------------------------------------------------------------------*/
// FILTERS
/*-----------------------------------------------------------------------------*/
// 0=no filtering, 1=lowpass, 2=wflc, 3=notch filter
typedef enum {
    ALLPASS,
    LOWPASS,
    HIGHPASS,
    BANDPASS,
    NOTCH,
} t_FILTER_TYPE;

/*-----------------------------------------------------------------------------*/
//  Complex data handling
/*-----------------------------------------------------------------------------*/
typedef struct {
    double real;
    double imag;
} t_COMPLEX;

/*-----------------------------------------------------------------------------*/
//  Function prototypes
/*-----------------------------------------------------------------------------*/
```

```
//typedef double (*fp_FILTER)(double datum, void *filter_para);

double filter (t_FILTER *f, double x);
void clear_filter (t_FILTER *f);
void init_lowpass_filter (t_FILTER *f, int order, double fc, double fs);
void init_highpass_filter (t_FILTER *f, int order, double fc, double fs);
void init_bandpass_filter (t_FILTER *f, int order, double fc1, double fc2, double fs);
void init_envelope_filter (t_FILTER *f, double fc, double fs);
double filter_envelope (t_FILTER *f, double x);
double filter_delay (t_FILTER *f, double x, int delay);
double filter_median (t_FILTER *f, double x);
void sort (double *arr, int beg, int end);
void swap (double *a, double *b);
int Construct_Butterworth_Filter (t_FILTER *f, int type, int order, double fc, double fs);

int poly_mult (t_POLYNOMIAL *A, t_POLYNOMIAL *B);
void poly_copy (t_POLYNOMIAL *A, t_POLYNOMIAL *B);

#endif // _FILTER_H_
//***************************************End filter.h **************************
```

A.4 READING INPUT PARAMETERS (READ_INI.C & READ_INI.H)

```
//***************************************Begin read_ini.c **************************
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "filter.h"
#include "filter_wflc.h"
#include "read_ini.h"

t_INI_PARAMETERS ini_para = {
    { "tremor_removal_type", 2, 2, "0(no filter), 1(lowpass), 2(wflc), 3(wflc modified)"},
    { "filter_order", 3, 3, "Filter order to be used (2 or 3)"},
    //[JOY_STICK]
    { "joyStick_idle_pos.X", 2050, 2050, "Joystick idle position X"},
    { "joyStick_idle_pos.Y", 2050, 2050, "Joystick idle position Y"},
    { "sampling_freq(Hz)", 100, 100, "sampling frequency"},
    //[PRE_FILTER]
    //# To filter out some glitches in input data, we may use a prefilter.
    { "prefilter_type", 2, 2, "0(No prefiltering), 1(low-pass), 2(median)"},
    { "prefilter.cut(Hz)", 8, 8, "cutoff frequency of low-pass prefilter"},
    { "pre_median_order", 3, 3, "order of median prefilter (# of data samples)"},
    //[LOW_PASS]
    { "intended_max_freq(Hz)", 2.5, 2.5, "intended movement max freq"},
    //[BAND_PASS] //# band-pass parameters are to control WFLC/FLC blocks
    { "tremor_freq_min(Hz)", 2.5, 2.5, "tremo minimum freq"},
    { "tremor_freq_max(Hz)", 8, 8, "tremo maximum freq"},
    { "alpha", 0, 0, "used by modified WFLC (removal_type 3) to control FLC's mu\n"
        "#\t\t# to control mu of FLC: mu' = mu * (1 + alpha*BW),\n"
        "#\t\t# BW = normalized tremo bandwidth at a moment (0 <= BW <= 1)"},
    { "", 0, 0, ""}}; // a null string keyword indicates the end of non-W/FLC parameters

/*----------------------------------------------------------------------------*/
//  Read an .ini file
/*----------------------------------------------------------------------------*/
int ReadIniFile (char *fname)
{
    int i, n;
    int ini_id;
    char buf[256];
    FILE *fin, *fout;

    t_INI_PARA *ini = (t_INI_PARA *)&ini_para;
    double mu, mub, mu0, f0, w1, wM1;
    int M;

    fin = fopen (fname, "r");
    if (! fin)
    {
       fout = fopen (fname, "w");
       fprintf (fout, "#\n"
               "# Filter parameters\n"
               "#\n"
               "# You can modify values after '=' sign as you wish.\n"
               "# You should not change the keyword before '=' sign.\n"
               "# (To comment out a line, add '#' or ';' at the first column.)\n#\n");
       for (i = 0; i < sizeof(t_INI_PARAMETERS)/sizeof(t_INI_PARA); i++)
       {
           if (*(ini[i].keyword) == 0) break;
           fprintf (fout, "%s=%g\t\t# %s\n", ini[i].keyword, ini[i].default_val,
ini[i].comment);
       }
       // Now for WFLC/FLC parameters
       fprintf (fout, "\n"
           "[WFLC]\n"
           "# WFLC parameters (to track tremor frequency)\n"
           "# Set mub to zero (or a very small value)\n"
           "#\n"
           "# mu      mub            m0             f0      M       w1      wM1\n"
           "0.0003    0.             0.0000002      4       1       0.      0.\n"
           "\n"
           "[FLC]\n"
           "# FLC parameters (to track actual intended signal)\n"
           "# 1. Set mu0 = 0 (or a very small value)\n"
           "# 2. f0 will be from WFLC.\n"
           "# mu value will be adjusted by tremor amplitude (envelope) by\n"
           "#   mu' = mu * (1 + alpha * tremor_bandwidth)\n"
           "# to control WFLC/FLC's the stop band width.\n"
           "#\n"
           "# mu      mub            m0             f0      M       w1      wM1\n"
           "0.02      0.00002        0.             4       1       0.      0.\n");
       fflush (fout);
       fclose(fout);
       fin = fopen (fname, "r");
    }
    if (! fin) return -1;   // fail to read ini file
    ini_id = 0;
    while (fgets(buf, 255, fin) != NULL) {
       if (buf[0] == '#' || buf[0] == ';' || buf[0] == '\n') continue;
       if (strncmp(buf, "[WFLC]", 6) == 0) {
           ini_id = 1;
       } else if (strncmp(buf, "[FLC]", 5) == 0) {
           ini_id = 2;
       } else if (ini_id == 0) {
           for (i = 0; *(ini[i].keyword) && i<sizeof(t_INI_PARAMETERS)/sizeof(t_INI_PARA); i++)
{
               n = strlen(ini[i].keyword);
               if (strncmp(buf, ini[i].keyword, n)) continue;
               sscanf (&buf[n], "=%lf", &ini[i].value);
               break;
           }
       } else {
           sscanf (buf, "%lf %lf %lf %lf %d %lf %lf", &mu, &mub, &mu0, &f0, &M, &w1, &wM1);
           ini_para.wflc[ini_id-1].mu = mu;
           ini_para.wflc[ini_id-1].mub = mub;
           ini_para.wflc[ini_id-1].mu0 = mu0;
           ini_para.wflc[ini_id-1].f0 = f0;
           ini_para.wflc[ini_id-1].M = M;
           ini_para.wflc[ini_id-1].w1 = w1;
```

```
            ini_para.wflc[ini_id-1].wM1 = wM1;
        }
    }
    fclose(fin);

#if 0
    // confirm ini read
    for (i = 0; i < sizeof(t_INI_PARAMETERS)/sizeof(t_INI_PARA); i++) {
       if (*(ini[i].keyword) == 0) break;
       printf ("%s=%g\t%g\n", ini[i].keyword, ini[i].default_val, ini[i].value);
    }
    for (i = 0; i < 2; i++)
    {
       printf ("WFLC[%d]: %11g %11g %11g %11g %6d %11g %11g\n", i,
           ini_para.wflc[i].mu,
           ini_para.wflc[i].mub,
           ini_para.wflc[i].mu0,
           ini_para.wflc[i].f0,
           ini_para.wflc[i].M,
           ini_para.wflc[i].w1,
           ini_para.wflc[i].wM1 );
    }
#endif
    return 0;
}
//***************************************End read_ini.c **************************

//***************************************Begin read_ini.h **************************
#ifndef _READ_INI_H_
#define _READ_INI_H_

typedef struct {
    char *keyword;
    double default_val;
    double value;
    char *comment;
} t_INI_PARA;

typedef struct {
    t_INI_PARA tremor_removal_type;
               //=2 # tremor_removal_type=0(no filter), 1(lowpass), 2(wflc)
    t_INI_PARA filter_order;          //=3 # (now fixed in the program)
    //[JOY_STICK]
    t_INI_PARA joystick_idle_pos_X;   //=2050 # joystick idle positions of X
    t_INI_PARA joystick_idle_pos_Y;   //=2050 # joystick idle positions of Y
    t_INI_PARA sampling_freq; //=100 # sampling rate 100 Hz

    //[PRE_FILTER]
    //# To filter out some glitches in input data, we may use a prefilter.
    t_INI_PARA prefilter_type;               //=2 # 0(No prefiltering), 1(low-pass), 2(median)
    t_INI_PARA prefilter_lpf_cutoff;  //=10 # cutoff frequency of low-pass prefilter
    t_INI_PARA prefilter_median_order;       //=3 # order of median prefilter (# of data samples)
    //[LOW_PASS]
    t_INI_PARA intended_frea_max;     //=2.5 # cutoff frequency of lowpass
(tremor_removal_type=1)

    //[BAND_PASS]
    //# band-pass parameters are to control WFLC/FLC blocks
    t_INI_PARA tremor_freq_min;       //=2.5 # lower cutoff frequency of bandpass for WFLC
    t_INI_PARA tremor_freq_max;       //=8 # upper cutoff frequency of bandpass for WFLC
    t_INI_PARA alpha;         //=0.   # to control mu of FLC: mu' = mu * (1 + alpha*BW),
                              //# BW = normalized tremo bandwidth at a moment (0 <= BW <= 1)
    t_INI_PARA empty;
    //[WFLC] and [FLC]
    struct {
       int M;
       double mu, mub, mu0, f0, w1, wM1;
    } wflc[2];
} t_INI_PARAMETERS;

int ReadIniFile (char *fname);
```

```
#endif // _READ_INI_H_
//***************************************End read_ini.h **************************
```

APPENDIX B

OUTPUT FILE

B.1 EXAMPLE OF OUTPUT FILE (VDD_OUT_001.DAT)

```
# No  Time   X     Y    Lowpass(X,Y)   WFLC(X,Y)  mod_WFLC(X,Y) ChairX ChairY Crashes  band(X) f0(tremor)  ARC VEL   mu'        TrackErr  Side
  0    0    2087  2034  1100  1100    2087  2034  2087  2034    100   325    0      56.814   4.000      0   0  0.0205681     0        1
  1    10   2088  2039  1100  1100    2086  2039  2086  2039    100   325    0     161.416   4.000      0   0  0.0218982     0        1
  2    20   2089  2035  1100  1100    2086  2035  2086  2036    100   325    0     243.445   3.999      0   0  0.0233836     0        1
  3    30   2088  2033  1100  1100    2084  2034  2083  2034    100   325    0     295.473   3.997      0   0  0.0246465     0        1
  4    40   2086  2030  1100  1100    2081  2031  2080  2031    100   325    0     313.593   3.993      0   0  0.0254592     0        1
  5    50   2088  2032  1100  1100    2083  2034  2082  2034    100   325    0     297.833   3.987      0   0  0.0258655     0        1
  6    60   2088  2037  1100  1100    2083  2039  2082  2039    100   325    0     251.744   3.979      0   0  0.0260687     0        1
  7    80   2089  2039  1100  1100    2084  2040  2083  2041    100   325    0     181.785   3.971      0   0  0.0261703     0        1
  8   100   2087  2037  1100  1100    2083  2038  2082  2038    100   325    0      96.540   3.967      0   0  0.0262211     0        1
  9   110   2087  2032  1100  1100    2084  2033  2082  2033    100   325    0       5.755   3.967      0   0  0.0262465     0        1
 10   120   2089  2031  1100  1100    2087  2031  2086  2032    100   325    0     -80.636   3.972      0   0  0.0262592     0        1
 11   140   2086  2031  1100  1100    2085  2031  2084  2031    100   325    0    -153.893   3.981      0   0  0.0262655     0        1
 12   160   2090  2033  1100  1100    2090  2032  2090  2033    100   325    0    -206.991   3.992      0   0  0.0262687     0        1
 13   170   2086  2036  1100  1100    2087  2035  2087  2035    100   325    0    -235.487   4.002      0   0  0.0262703     0        1
 14   190   2089  2040  1164  1134    2091  2038  2091  2039    100   325    0    -237.757   4.008      0   0  0.0262711     0        1
 15   200   2089  2039  1293  1260    2092  2037  2092  2037    100   325    0    -214.882   4.009      0   0  0.0262715     0        1
 16   220   2087  2036  1417  1381    2091  2033  2091  2033    100   325    0    -170.768   4.006      0   0  0.0262717     0        1
 17   230   2088  2031  1535  1495    2092  2028  2093  2028    100   325    0    -111.170   4.002      0   0  0.0261142     0        1
 18   240   2088  2031  1645  1603    2093  2028  2093  2028    100   325    0     -43.129   4.000      0   0  0.0255745     0        1
 19   260   2090  2031  1747  1702    2094  2028  2095  2028    100   325    0      25.786   4.003      0   0  0.0251648     0        1
 20   280   2090  2032  1839  1792    2094  2030  2095  2029    100   325    0      88.299   4.012      0   0  0.02496       0        1
 21   290   2087  2032  1922  1873    2090  2030  2092  2030    100   325    0     138.064   4.025      0   0  0.0248576     0        1
 22   300   2087  2033  1995  1944    2090  2031  2091  2031    100   325    0     170.557   4.042      0   0  0.0248063     0        1
 23   310   2089  2035  2059  2006    2091  2034  2092  2033    100   325    0     183.444   4.057      0   0  0.0247807     0        1
 24   340   2087  2035  2112  2058    2088  2034  2089  2034    100   325    0     176.387   4.068      0   0  0.0247679     0        1
 25   350   2091  2030  2156  2101    2091  2029  2092  2029    100   325    0     151.268   4.075      0   0  0.0247615     0        1
 26   370   2088  2033  2192  2135    2087  2033  2088  2033    100   325    0     111.676   4.077      0   0  0.0247583     0        1
 27   390   2088  2037  2219  2161    2086  2037  2087  2037    100   325    0      62.443   4.076      0   0  0.024528      0        1
 28   410   2087  2031  2238  2180    2085  2031  2085  2031    100   325    0       9.314   4.077      0   0  0.0240984     0        1
 29   420   2088  2034  2251  2193    2085  2034  2086  2034    100   325    0     -41.899   4.080      0   0  0.0238837     0        1
 30   440   2088  2036  2258  2199    2085  2036  2085  2036    100   325    0     -85.902   4.089      0   0  0.0237763     0        1
```

BIBLIOGRAPHY

[1] D. Gillard, T. Cameron, A. Prochazka, and M. Gauthier, “Tremor suppression using functional electrical stimulation: a comparison between digital and analog controllers,” *IEEE Trans. Rehab. Eng.,* vol. 7, no. 3, pp. 385-388, 1999.

[2] R. Eberhart and X. Hu, “Human tremor analysis using particle swarm optimization,” *Proc. IEEE*, pp.1927-1930, 1999.

[3] P. Riley and M. Rosen, “Evaluating manual control devices for those with tremor disability,” *J. Rehab. Res. Develop.*, vol. 24, no. 2, pp. 99-110, 1987.

[4] R. Cooper, D. Spaeth, D. Jones, M. Boninger, S. Fitzgerald, S. Guo, “Technical Note: Comparison of virtual and real electric powered wheelchair driving using a position sensing joystick and an isometric joystick,” Medical Engineering and Physics, vol. 24, no. 10, pp. 703-708, 2002.

[5] M. Amin, “Interference mitigation in spread spectrum communication systems using time-frequency distributions,” *IEEE Trans. Sig. Process.*,, vol. 45, no. 1, pp. 90-101, 1997.

[6] B. Choi, W. Nho, T. Liu, and G. Salama, “Life Span of Ventricular Fibrillation Frequencies,” *Circulation Research*, vol. 91(4), pp. 339-345, August 23, 2002.

[7] J. Gonzalez, E. Heredia, and T. Rahman, Optimal digital filtering for tremor suppression,*” IEEE Trans. Biomedical Eng.*, vol.47, pp. 664-673, May 2000.

[8] S. Pledgie, K. Barner, S. Agrawal, and T. Rahman, “Tremor suppression through impedance control,” *IEEE Trans. Rehab. Eng.*, vol. 8, no. 1, pp. 53-59, 2000.

[9] R. Rao, R. Seliktar, and T. Rahman, “Evaluation of an isometric and position joystick in a target acquisition task for individuals with cerebral palsy,” *IEEE Trans. Rehab. Eng.*, vol. 8, no. 1, pp. 118-125, 2000.

[10] S. Pledgie, K. Barner, S. Agrawal, T. Rahman, “Tremor suppression through impedance control,” *IEEE Trans. Rehabilitation Eng.*, vol. 8, no. 1, pp.53-59, March 2000.

[11] D. Hsu, W. Huang, and N. Thakor, “On-line adaptive canceling of pathological tremor for computer pen handwriting,” *Proc. IEEE*, pp. 113-114, 1995.

[12] J. Aylor, R. Ramey, J. Schaddegg and S. Reger, "Versatile wheelchair control system," *Medical and Biological Engineering and Computing*, vol. 17, pp. 110-114, 1979.

[13] R. Edwards and A. Beuter, "Indexes for identification of abnormal tremor using computer tremor evaluation systems," *IEEE Trans. Biomedical Eng.*, vol.46, no. 7, pp. 895-898, July 1999.

[14] B. Zwaag and L. Jain, "Minimizing tremor in a joystick controller using Fuzzy logic," *Conference on Knowledge-Based Intelligent Information Engineering Systems*, Adelaide, Australia, pp. 5-8, 1999.

[15] C. Riviere, R. Scott Rader, and N. Thakor, "Adaptive canceling of physiological tremor for improved precision in microsurgery," *IEEE Trans. Biomedical Eng.*, vol. 45, no. 7, pp. 839-846, July 1998.

[16] C. Avizzano, A. Brogni, and M. Bergamasco, "Application of filtering stratergies to multiple sclerosis tremor: analysis of results," ?

[17] L. Cohen, *Time-Frequency Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[18] L. Cohen and C. Lee, "Instantaneous frequency, its standard deviation, and multicomponent signals," *Proc. SPIE*, vol.975, pp. 186-208, 1988.

[19] D. Gabor, "Theory of communication," *J. Inst. Elect. Eng.*, vol. 93, pp. 429-457, 1946.

[20] W. Nho and P. Loughlin, "When is instantaneous frequency the average frequency at each time?," *IEEE Signal Processing Lett.*, vol. 6, pp.78-80, Apr. 1999.

[21] N-C. Huang and J. Aggarwal, "Frequency-domain considerations of LTV digital filters," *IEEE Trans. Circuits and Syst*., vol. 28, no. 4, pp. 279-287, 1981.

[22] J. Proakis and D. Manolakis, *Digital Signal Processing-Principles, Algorithms, and Applications*, Macmillan, 1992.

[23] S. Orfanidis, *Introduction to Signal Processing*, Prentice-Hall, 1995.

[24] B. Boashash, "Estimating and interpreting the instantaneous frequency of a signal – part 2: Algorithms and applications," *Proceedings of the IEEE*, vol. 80, pp. 539-568, Apr. 1992.

[25] L. Griffiths, "Rapid measurement of digital instantaneous frequency," *IEEE Trans. Acoust. Speech, Signal Processing*, pp.202-221, 1975.

[26] S. Haykin, Adaptive Filter Theory. Englewood Cliffs, NJ, Prentic Hall, 2nd ed., 1991.

[27] D. Brienza and J. Angelo, "A force feedback joystick and control algorithm for wheelchair obstacle avoidance," *Disability and Rehabilitation*, vol. 18, no. 3, pp. 123-129, 1996.

[28] D. Spaeth, Evaluation of an Isometric joystick with Control Enhancing Algorithms for Improved Driving of Electric Powered Wheelchair, Ph.D. Thesis, University of Pittsburgh, 2002.

[29] R. Stiles and J. Randall, Mechanical factors in human tremor frequency, J. Applied Physology, 1967.

[30] C. Vaz, X. Kong, and N. Thakor, “An adaptive estimation of periodic signals using a Fourier linear combiner,” IEEE Trans. Signal Processing, vol. 42, pp. 1-10, Jan. 1994.